

21st Large Installation System Administration Conference

Dallas, TX, USA

November 11–16, 2007

Sponsored by

The **USENIX Association** and **SAGE**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

THE USENIX SIG FOR

[sage]
SYSADMINS

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: +1 510-528-8649
<http://www.usenix.org>
<office@usenix.org>

Past LISA Conferences

LISA '06: 20th Large Installation System Administration Conference	Dec. 3-8, 2006	Washington, DC
LISA '05: 19th Large Installation System Administration Conference	Dec. 4-9, 2005	San Diego, CA
LISA '04: 18th Large Installation System Administration Conference	Nov. 14-19, 2004	Atlanta, GA
LISA '03: 17th Large Installation Systems Administration Conference	Oct. 26-31, 2003	San Diego, CA
LISA '02: 16th Systems Administration Conference	Nov. 3-8, 2002	Philadelphia, PA
LISA 2001: 15th Systems Administration Conference	Dec. 2-7, 2001	San Diego, CA
LISA 2000: 14th Systems Administration Conference	Dec. 3-8, 2000	New Orleans, LA
LISA '99: 13th Systems Administration Conference	Nov. 7-12, 1999	Seattle, WA
LISA '98: 12th Systems Administration Conference	Dec. 6-11, 1998	Boston, MA
LISA '97: 11th Systems Administration Conference	Oct. 26-31, 1997	San Diego, CA
LISA '96: 10th System Administration Conference	Sept. 29-Oct. 4, 1996	Chicago, IL
LISA '95: 9th System Administration Conference	Sept. 18-22, 1995	Monterey, CA
LISA '94: 8th USENIX System Administration Conference	Sept. 19-23, 1994	San Diego, CA
LISA '93: USENIX 7th System Administration Conference	Nov. 1-5, 1993	Monterey, CA
LISA VI: 6th Systems Administration Conference	Oct. 19-23, 1992	Long Beach, CA
5th Large Installation Systems Administration Conference	Sept. 30-Oct. 3, 1991	San Diego, CA
4th Large Installation System Administrator's Conference	Oct. 17-19, 1990	Colorado Springs, CO
Workshop on Large Installation Systems Administration III	Sept. 7-8, 1989	Austin, TX
Workshop on Large Installation Systems Administration	Nov. 17-18, 1988	Monterey, CA
Workshop on Large Installation Systems Administration	April 9-10, 1987	Philadelphia, PA

Copyright © 2007 by The USENIX Association. All rights reserved.
This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.
Permission is granted for the noncommercial reproduction
of the complete work for educational or research purposes.

USENIX acknowledges all trademarks appearing herein.

ISBN-13: 978-1-931971-55-3
ISBN-10: 1-931971-55-2

USENIX Association

**Proceedings of the
21st Large Installation
System Administration Conference
(LISA '07)**

**November 11-16, 2007
Dallas, TX, USA**

ACKNOWLEDGMENTS

PROGRAM CHAIR

Paul Anderson, *University of Edinburgh*

PROGRAM COMMITTEE

Charlie Catlett, *NSF Teragrid Project*
 William Cheswick, *Consultant, cheswick.com*
 Alva Couch, *Tufts University*
 Eileen Frisch, *Exponential Consulting*
 Peter Baer Galvin, *Corporate Technologies, Inc.*
 Andrew Hume, *AT&T Labs – Research*
 William LeFebvre, *Independent Consultant*
 Adam Moskowitz, *Constant Contact*
 Sanjai Narain, *Telcordia Technologies*
 Melanie Rieback, *Vrije Universiteit Amsterdam*
 Kent Skaar, *Bladelogic*
 Chad Verbowski, *Microsoft Research*

INVITED TALKS COORDINATORS

Rudi van Drunen, *Competa IT/Xlexit*
 Doug Hughes, *D. E. Shaw Research, LLC*

WORKSHOPS COORDINATOR

Lee Damon, *University of Washington*

GURU IS IN COORDINATORS

Philip Kizer, *Estacado Systems*
 John “Rowan” Littell, *California College of the Arts*

HIT THE GROUND RUNNING TRACK COORDINATOR

Adam Moskowitz, *Constant Contact*

WORK-IN-PROGRESS REPORTS/POSTERS COORDINATOR

Brent Hoon Kang, *University of North Carolina at Charlotte*

FIGHTING SPAM WORKSHOP

Chris St. Pierre, *Nebraska Wesleyan University*

SERVER ROOM BEST PRACTICES WORKSHOP

Hunter Matthews, *Duke University*

MICROLISA WORKSHOP

Robert Au

UNIVERSITY ISSUES WORKSHOP

John “Rowan” Littell, *California College of the Arts*

CONFIGURATION WORKSHOP

Mark Burgess, *Oslo University College*
 Sanjai Narain, *Telcordia Technologies, Inc.*

SYSTEM ADMINISTRATION EDUCATION WORKSHOP

Brent Hoon Kang, *University of North Carolina at Charlotte*
 Alva Couch, *Tufts University*
 Mark Burgess, *Oslo University College*

ADVANCED TOPICS WORKSHOP

Adam Moskowitz, *Constant Contact*

PROCEEDINGS TYPESETTING

Rob Kolstad

CONFERENCE ORGANIZATION

The USENIX Association Staff

EXTERNAL REVIEWERS

Tim Colles
 Jon Finke
 Heather Flanagan
 David Harnick-Shapiro
 David Hoffman
 Tim Hunter
 Duncun Hutty
 Brent Kang
 Scott Karlin
 Digant Kasundra
 Philip Kizer
 Juhan Lee

John “Rowan” Littell
 Steve Loughran
 Kenny MacDonald
 Sean MacGeever
 German Cancio Melia
 Lee Merrick
 Mario Obejas
 Cat Okita
 Kathryn Penn
 Stephen Quinney
 Sanjay Rao
 Amy Rich

George Ross
 John Sellens
 Ed Smith
 Marc Staveley
 Andrew Stribblehill
 Leon Towns-von Stauber
 Rudi van Drunen
 Simon Wilkinson
 David Williamson
 Alan Wilson
 Pat Wilson
 Alexios Zavras

CONTENTS

Acknowledgments	ii
Index of Authors	v
Message from the Program Chair	vii

WEDNESDAY, NOVEMBER 14

Opening Remarks, Awards, and Keynote Session Chair: Paul Anderson

Autonomic Administration: HAL 9000 Meets Gene Roddenberry

John Strassner – Motorola

Security via Firewalls Session Chair: Aileen Frisch

1 PolicyVis: Firewall Security Policy Visualization and Inspection
Tung Tran, Ehab Al-Shaer, and Raouf Boutaba – University of Waterloo, Canada

17 Inferring Higher Level Policies from Firewall Rules
Alok Tongaonkar, Niranjan Inamdar, and R. Sekar – Stony Brook University

27 Assisted Firewall Policy Repair Using Examples and History
Robert Marmorstein and Phil Kearns – The College of William & Mary

Performance Session Chair: Melanie Rieback

39 NetADHICT: A Tool for Understanding Network Traffic
Hajime Inoue – ATC-NY, Ithaca, NY; Dana Jansens, Abdulrahman Hijazi, and Anil Somayaji – Carleton University, Ottawa, Canada

49 CAMP: A Common API for Measuring Performance
Mark Gabel and Michael Haungs – California Polytechnic State University, San Luis Obispo

63 Application Buffer-Cache Management for Performance: Running the World's Largest MRTG
David Plonka, Archit Gupta, and Dale Carder – University of Wisconsin-Madison

Virtualization Session Chair: Alva Couch

79 Stork: Package Management for Distributed VM Environments
Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John H. Hartman – University of Arizona

95 Decision Support for Virtual Machine Re-Provisioning in Production Environments
Kyrre Begnum and Matthew Disney – Oslo University College, Norway; Aileen Frisch – Exponential Consulting; Ingard Mevåg – Oslo University College

105 OS Circular: Internet Client for Reference
Kuniyasu Suzuki, Toshiki Yagi, Kengo Iijima, and Nguyen Anh Quynh – National Institute of Advanced Industrial Science and Technology, Japan

117 Secure Isolation of Untrusted Legacy Applications
Shaya Potter, Jason Nieh, and Matt Selsky – Columbia University

THURSDAY, NOVEMBER 15

Miscellaneous Topics, I

Session Chair: Sanjai Narain

- 131 Policy-Driven Management of Data Sets**
Jim Holl, Kostadis Roussos, and Jim Voll – Network Appliance, Inc.
- 141 ATLANTIDES: An Architecture for Alert Verification in Network Intrusion Detection Systems**
Damiano Bolzoni – University of Twente, The Netherlands; Bruno Crispo – Vrije Universiteit, The Netherlands & University of Trento, Italy; Sandro Etalle – University of Twente, The Netherlands
- 153 PDA: A Tool for Automated Problem Determination**
Hai Huang, Raymond Jennings III, Yaoping Ruan, Ramendra Sahoo, Sambit Sahu, and Anees Shaikh – IBM T. J. Watson Research Center

Managing Grids and Clusters

Session Chair: Chad Verbowski

- 167 Usher: An Extensible Framework for Managing Clusters of Virtual Machines**
Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker – University of California, San Diego
- 183 Remote Control: Distributed Application Configuration, Management, and Visualization with Plush**
Jeannie Albrecht – Williams College; Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat – University of California, San Diego
- 203 Everlab – A Production Platform for Research in Network Experimentation and Computation**
Elliot Jaffe, Danny Bickson, and Scott Kirkpatrick – Hebrew University of Jerusalem, Israel

Miscellaneous Topics, II

Session Chair: Alva Couch

- 215 Master Education Programmes in Network and System Administration**
Mark Burgess – Oslo University College; Karst Koymans – Universiteit van Amsterdam
- 231 On Designing and Deploying Internet-Scale Services**
James Hamilton – Windows Live Services Platform
- 243 RepuScore: Collaborative Reputation Management Framework for Email Infrastructure**
Gautam Singaraju and Brent ByungHoon Kang – University of North Carolina at Charlotte

FRIDAY, NOVEMBER 16

Configuration Management

Session Chair: Paul Anderson

- 253 Moobi: A Thin Server Management System Using BitTorrent**
Chris McEniry – Sony Computer Entertainment America
- 261 PoDIM: A Language for High-Level Configuration Management**
Thomas Delaet and Wouter Joosen – Katholieke Universiteit Leuven, Belgium
- 275 Network Patterns in Cfengine and Scalable Data Aggregation**
Mark Burgess and Matthew Disney – Oslo University College; Rolf Stadler – KTH Royal Institute of Technology, Stockholm

INDEX OF AUTHORS

Ehab Al-Shaer	1	Wouter Joosen	261
Jeannie Albrecht	183	Brent ByungHoon Kang	243
Scott Baker	79	Phil Kearns	27
Kyrre Begnum	95	Scott Kirkpatrick	203
Danny Bickson	203	Karst Koymans	215
Damiano Bolzoni	141	Robert Marmorstein	27
Matt Borgard	79	Chris McEniry	253
Raouf Boutaba	1	Marvin McNett	167
Ryan Braud	183	Ingard Mevåg	95
Mark Burgess	215, 275	Jason Nieh	117
Justin Cappos	79	Duy Nyugen	79
Dale Carder	63	Jeremy Plichta	79
Bruno Crispo	141	David Plonka	63
Darren Dao	183	Shaya Potter	117
Thomas Delaet	261	Nguyen Anh Quynh	105
Matthew Disney	95, 275	Kostadis Roussos	131
Sandro Etalle	141	Yaoping Ruan	153
Æleen Frisch	95	Ramendra Sahoo	153
Mark Gabel	49	Sambit Sahu	153
Archit Gupta	63	R. Sekar	17
Diwaker Gupta	167	Matt Selsky	117
James Hamilton	231	Anees Shaikh	153
Jason Hardies	79	Gautam Singaraju	243
John H. Hartman	79	Alex C. Snoeren	183
Michael Haungs	49	Anil Somayaji	39
Abdulrahman Hijazi	39	Rolf Stadler	275
Jim Holl	131	Kuniyasu Suzaki	105
Hai Huang	153	Alok Tongaonkar	17
Kengo Iijima	105	Nikolay Topilski	183
Niranjan Inamdar	17	Tung Tran	1
Hajime Inoue	39	Christopher Tuttle	183
Elliot Jaffe	203	Amin Vahdat	167, 183
Dana Jansens	39	Geoffrey M. Voelker	167
Raymond Jennings III	153	Jim Voll	131
Jeffrey Johnston	79	Toshiki Yagi	105

Message from the Program Chair

Dear Reader:

For twenty years, the annual LISA conference has been the foremost worldwide gathering for everyone interested in the technical and administrative issues of running a large computing facility. We are now being expected to manage installations of increasing size and complexity, with a higher degree of reliability and performance; I believe that this will only be possible through collaboration among different disciplines, from practising administrators with many years experience, to academics with new insights and approaches. LISA is unique in bringing together this wide range of people to address real-world problems.

These Proceedings contain the complete text of the refereed papers presented at the 2007 LISA conference. The 22 papers were selected from a total of 55 submissions, and I would hope that practising system administrators will find most of these to be relevant and accessible – the papers include “hot” new topics, such as the management of virtual machines, and new perspectives on old problems such as spam and firewall configuration. They range from “soft” topics such as system administrator education, to more technical issues such as configuration management.

Putting together the LISA conference has required the energy and experience of so many people, and I would like to thank all those who have been involved. I hope that you find the contents of these proceedings useful and inspiring, and that you may consider submitting your own work to a future LISA conference.

Paul Anderson
Program Chair
LISA '07

PolicyVis: Firewall Security Policy Visualization and Inspection

Tung Tran, Ehab Al-Shaer, and Raouf Boutaba – University of Waterloo, Canada

ABSTRACT

Firewalls have an important role in network security. However, managing firewall policies is an extremely complex task because the large number of interacting rules in single or distributed firewalls significantly increases the possibility of policy misconfiguration and network vulnerabilities. Moreover, due to low-level representation of firewall rules, the semantic of firewall policies become very incomprehensible, which makes inspecting of firewall policy's properties a difficult and error-prone task.

In this paper, we propose a tool called PolicyVis which visualizes firewall rules and policies in such a way that efficiently enhances the understanding and inspecting firewall policies. Unlike previous works that attempt to validate or inspect firewall rules based on specific queries or errors, our approach is to visualize firewall policies to enable the user to place general inquiry such as "does my policy do what I intend to do" unrestrictedly. We describe the design principals in PolicyVis and provide concepts and examples dealing with firewall policy's properties, rule anomalies and distributed firewalls. As a result, PolicyVis considerably simplifies the management of firewall policies and hence effectively improves the network security.

Introduction

Network security is essential to the development of internet and has attracted much attention in research and industrial communities. With the increase of network attack threats, firewalls are considered effective network barriers and have become important elements not only in enterprise networks but also in small-size and home networks. A firewall is a program or a hardware device to protect a network or a computer system by filtering out unwanted network traffic. The filtering decision is based on a set of ordered filtering rules written based on predefined security policy requirements.

Firewalls can be deployed to secure one network from another. However, firewalls can be significantly ineffective in protecting networks if policies are not managed correctly and efficiently. It is very crucial to have policy management techniques and tools that users can use to examine, refine and verify the correctness of written firewall filtering rules in order to increase the effectiveness of firewall security.

It is true that humans are well adapted to capture data essences and patterns when presented in a way that is visually appealing. This truth promotes visualization on data, on which the analysis is very hard or ineffective to carry out because of its huge volume and complexity. The amount of data that can be processed and analyzed has never been greater, and continues to grow rapidly. As the number of filtering rules increases largely and the policy becomes much more complex, firewall policy visualization is an indispensable solution to policy management. Firewall policy visualization helps users understand their policies easily and grasp complicated rule patterns and behaviors efficiently.

In this paper, we present PolicyVis, a useful tool in visualizing firewall policies. We describe design principals, implementations and application examples that deal with discovering firewall policy's properties, rule anomalies for single or distributed firewalls.

Although network security visualization has been given strong attention in the research community, the emphasis was mostly on the network traffic [4, 8]. On the other hand, tools in [12, 16] visualize some firewall aspects, but don't give users a thorough look at firewall policies.

This paper is organized as follows. In the next section, we summarize related work. We then describe PolicyVis design principals followed by descriptions of scenarios that show the usefulness of PolicyVis. Next, we show how rule anomalies are visualized by PolicyVis and demonstrate some examples of determining rule anomalies by using PolicyVis. We then describe visualization on distributed firewalls in PolicyVis followed by a discussion of the implementation and evaluation of PolicyVis. Finally, we show conclusions and plans for future work.

Related Work

A significant amount of work has been reported in the area of firewall and policy-based security management. In this section, we focus our study on the work that closely related to PolicyVis' design objectives: network security visualization and policy management.

There are many visualization tools introduced to enhance network security. PortVis [15] uses port-based detection of security activities and visualizes network

traffic by choosing important features for axes and displaying network activities on the graph. SeeNet [3] supports three static network displays: two of these use geographical relationships, while the third is a matrix arrangement that gives equal emphasis to all network links. NVisionIP [11] uses a graphical representation of a class-B network to allow users to quickly visualize the current state of networks. Le Malécot, et al. [14] introduced an original visualization design which combines 2-D and 3-D visualization for network traffic monitoring. However, these tools only focus on visualizing network traffic to assist users in understanding network events and taking according actions.

Moreover, previous work on firewall visualization only concentrates on visualizing how firewalls react to network traffic based on network events. Chris P. Lee, et al. [12] proposed a tool visualizing firewall reactions to network traffic to aid users in configuration of firewalls. FireViz [16] visually displays activities of a personal firewall in real time to possibly find any potential loop holes in the firewall's security policies. These tools can only detect a small subset of all firewall behaviors and can not determine all possible potential firewall patterns by looking at the policy directly like PolicyVis. Besides, Tufin SecureTrack [19] is a commercial firewall operations management solution, however, it provides change management and version control for firewall policy update. It basically visualizes firewall policy version changes, but not rule properties and relations and allows users to receive alerts if any change violates the organizational policy. Thus, Tufin SecureTrack can not be used for rules analysis and anomaly discovery.

In the field of firewall policy management, a filtering policy translation tool proposed in [2] describes, in a natural textual language, the meaning and the interactions of all filtering rules in the policy, revealing the complete semantics of the policy in a very concise fashion. However, this tool is not as efficient as PolicyVis in helping users capture the policy properties quickly in case of huge number of rules in the policy. In [1], the authors mentioned firewall policy anomalies and techniques to discover them, and suggested a tool called Firewall Policy Advisor which implements anomaly discovery algorithms. However, Firewall Policy Advisor is not capable of showing all potential behaviors of firewall policies and doesn't help users in telling if a policy does what he wants.

The authors in [6, 10] suggest methods for detecting and resolving conflicts among general packet filters. However, only correlation anomaly [1] is considered because it causes ambiguity in packet classifiers. In addition, the authors in [13, 18] proposed firewall analysis tools that allow users to issue customized queries on a set of filtering rules and display corresponding outputs in the policy. However, the query reply could be overwhelming and still complex to understand. PolicyVis output is more comprehensible.

Moreover, those tools require users to consider very specific issues to inspect the policy. PolicyVis, on the other hand, enables users to investigate the policy at once, which is more practical and efficient in large policies.

PolicyVis Objectives

Information visualization is always an effective way to capture and explore large amount of abstract data. With the necessity of guaranteeing a correct firewall behavior, users need to recognize and fix firewall misconfigurations in a swift manner. However, the complexity of dealing with firewall policies is they are attributed to the large number of rules, rules complexity and rules dependency. Those facts motivate a tool which visualizes all firewall rules in such a way that rule interactions are easily grasped and analyzed in order to come up with an opportune solution to any firewall security breach.

PolicyVis is a visualization tool of firewall policies helps users to achieve the following goals in an effective and fast fashion:

- **Visualizing rule conditions, address space and action:** a firewall policy is attributed by rules format, rules dependency and matching semantics. Comprehensive visualization of firewall policies requires a mechanism of transforming firewall rules to visual elements which significantly enhance the investigation of policies. PolicyVis effectively visualizes all firewall rule core elements: conditions, address space and action.
- **Firewall policy semantic discovery:** it is a very normal demand of users to know all possible behaviors of a firewall to its intended protected system. With advantages of visualization and many graphic options supported by PolicyVis, all potential firewall behaviors can be easily discovered, which are normally very hard to grasp in a usual context.
- **Firewall policy rule conflict discovery:** PolicyVis can be able to not only give users a view on normal rule interactions, but also pinpoint all possible rule anomalies in the policy. This is a crucial feature of PolicyVis to become a very helpful tool for users. All kind of rule conflicts can be efficiently visualized without worrying about running any algorithm to find potential rule conflicts.
- **Firewall policy inspection based on users' intention:** with a policy of thousands of rules, it is much likely that the user will make configuration mistakes (not rule conflicts mentioned above) in the policy which causes the firewall to function incorrectly. PolicyVis brings all firewall rules to a graphic view so that all configuration mistakes are highlighted without any difficulty.
- **Visualizing distributed firewalls:** distributed firewalls security is as important as a single firewall, besides visualizing a single firewall.

PolicyVis also lets users visualize distributed firewalls with the same efficiency in all goals mentioned above.

PolicyVis Design Principles

The fundamental design requirements for PolicyVis included:

- **Simplicity:** It should be fairly intuitive for users to inspect firewall policies in a 2D graph using multiple fields. We chose to compress firewall rules into 2D graph with three factors because it is much likely that a certain field (like source port) can be ignored or not important when investigating the policy. 2D graph is simple but quite effective in terms of helping users thoroughly look into the policy's behaviors.
- **Expressiveness:** It is very important that users can easily capture true rule interactions so that appropriate actions can be taken immediately. PolicyVis supports very detailed and thorough visualization of all possible firewall rules' behaviors by considering all rules fields, rule orders as well as all rule actions.
- **Flexible Visualization scope:** PolicyVis allows users to visualize what they are interested in (the target: any rule field) so that all possible aspects of the policy can be viewed and analyzed. Moreover, with multiple dimensions support, PolicyVis is flexible in letting users choose desired fields for the graph coordinates, which is convenient and effective to observe and investigate the policy from different views. Besides, there are choices on type of traffic (accepted, denied or both) which can be viewed separately to meet users' different purposes.
- **Ability to Compress, Focus and Zoom:** It is a normal thing to take a closer look at a specific set of rules when investigating the policy. PolicyVis supports zooming so that users can closely investigate a set of considered rules. This zooming feature is very useful if too many rules get involved in the investigation and the axes get crowded. In addition, PolicyVis gives users the ability to investigate rule anomalies existing in the policy through the focusing feature. With PolicyVis, users can also visualize the whole policy at once as well as portions of the policy partitioned by ranges of a specific field. This is a helpful feature of PolicyVis when users want to consider the policy applied to a subnet or a desired portion of the network.
- **Ability to use policy segmentation:** In order to investigate accepted or denied traffic only, policy segmentation with BDDs technique [5] is a powerful means employed by PolicyVis to increase the effectiveness and correctness of extracting useful information from the policy.
- **Ability to use symbols, colors, notations:** Policies are attributed by rules format, rules

dependency and matching semantics (rule order). Moreover, firewall rules contain conditions (protocol, port and address), values (specific and wildcard) and actions (allowed and denied). PolicyVis visualizes those features using colors, symbols, and notations which are essential for users to capture quickly and easily the inside interactions and performance of firewall policies.

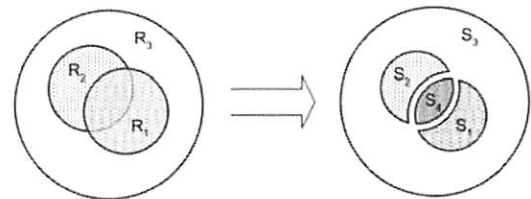
Multi-level Visualizing of Firewall Policies

Using PolicyVis, multi-level visualizing of firewall policies can be accomplished effectively. With PolicyVis' many flexible features, user can inspect the firewall policy from different views (like port level, address level, etc.) to understand all potential inside behaviors of the policy. In order to achieve this goal, PolicyVis deploys many methods and techniques which efficiently bring firewall policies to expressive visual views.

Using BDDs To Segment Policy and Find Accepted and Denied Spaces

Firewall policy segmentation using Binary Decision Diagram or BDD was first introduced by our group in [5, 9] to enhance the firewall validation and testing procedures. As defined in [5], a segment is a subset of the total traffic address space that belongs to one or more rules, such that each of the member elements (i.e., header tuples) conforms to exactly the same set of policy rules. Rules and address spaces are represented as Boolean expressions and BDD is used as a standard canonical method to represent Boolean expressions. By taking advantages of BDD's properties, firewall rules are effectively segmented into disjoint segments each of which belong to either accepted or denied space.

R1:	tcp	121.63.*.*: any	143.91.78.*: any	accept
R2:	tcp	121.63.71.*: any	143.91.*.*: any	accept
R3:	any	*.*.*.*: any	*.*.*.*: any	deny



(a) Policy address-space (b) Segmented address-space

Table 1: Example of firewall policy segmentation.

In specific, the authors in [9] suggest constructing a Boolean expression for a policy P_a using the rule constraints as follows:

$$P_a = \bigvee_{i \in \text{index}(a)} (\neg C_1 \wedge \neg C_2 \wedge \dots \wedge \neg C_{i-1} \wedge \neg C_i)$$

where $\text{index}(a)$ is the set of indices of rules that have a as their action and C_i is the rule condition of conjunctive fields. In other words,

$$\text{index}(a) = \{ i \mid R_i = C_i \leadsto a \}$$

This formula can be understood as saying that a packet will trigger action a if it satisfies the condition C_i for some rule R_i with action a , provided that the packet does not match the condition of any prior rule in the policy. Table 1 shows an example of a policy of three intersecting rules forming total of four independent segments of policy address space.

PolicyVis allows users to visualize only accepted or denied traffic; therefore it is important to efficiently extract those spaces from the policy. A naïve algorithm to achieve this might take exponential running time. Fortunately, policy segmentation using BDD is quite effective in doing this. We decided to employ BDDs for segmenting rules to quickly retrieve correct accepted and denied spaces. This makes PolicyVis a reliable and fast tool. Policy rules are segmented using BDD right after they are read from the input file. This ahead-of-time rule segmentation speed up the process when the user chooses to visualize only accepted or denied traffic.

Firewall Visualization Techniques

In this section, we describe visualization techniques and methods used in our PolicyVis tool to achieve the objectives. More specific techniques and algorithms to visualize firewall anomalies are described later.

To achieve the visualization effectiveness, PolicyVis supports both policy segments and policy rules visualization, which depends on properties of the

policy users want to examine. When dealing with only accepted or denied space, PolicyVis visualizes policy segments obtained from using BDD as mentioned earlier. However, when users choose to investigate both accepted and denied spaces together, PolicyVis visualizes policy rules because the union of both spaces returns to the original rules. Moreover, visualizing policy rules in this case helps users capture all possible rule interactions which is hard to conceive by looking at separate visualizations of both spaces.

When users investigate a firewall policy scope (a field and a value), PolicyVis collects all rules (or segments) that have the corresponding field as a superset of the scope input and visualizes those rules (or segments). When choosing a scope to investigate, users want to inspect how the firewall policy applies to that scope, thus rules (or segments) that include only the address space of the target scope. Rules (or segments) are represented as rectangles with different colors to illustrate different kinds of traffic (accepted or denied). Those colors are set transparent so that rules overlapping with the same or different actions can be effectively recognized. Moreover, different symbols (small square and circle) placed at the corner of rectangles are used for different traffic protocols (e.g., TCP, UDP, ICMP, IGMP) and notations (i.e., tooltips or legends) are used to determine rules' order and related things.

When multiple rectangles (rules or segments) are sketched from the same coordinates, colors and symbols might not be enough to tell what kind of traffic or

Order	Prot	SrcIP	SrcPort	DestIP	DestPort	Action
1	tcp	140.192.37.2	*	161.120.33.*	1433	ACCEPT
2	tcp	140.192.37.1	*	161.120.33.41	22	DENY
3	tcp	140.192.37.*	*	161.120.33.41	22	ACCEPT
4	tcp	140.192.37.4	*	161.120.33.44	1433	ACCEPT
5	tcp	140.192.37.5	*	161.120.33.44	1433	ACCEPT
6	tcp	140.192.37.*	*	161.120.33.44	1433	DENY
7	tcp	140.192.38.3	*	161.120.33.44	22	DENY
8	tcp	140.192.38.8	*	161.120.33.44	22	DENY
9	tcp	140.192.38.*	*	161.120.33.44	22	ACCEPT
10	tcp	140.192.36.2	*	161.120.34.45	22	DENY
11	tcp	140.192.36.*	*	161.120.34.45	22	ACCEPT
12	tcp	141.192.36.*	*	161.121.33.*	23	DENY
13	tcp	141.192.*.*	*	161.121.33.*	23	ACCEPT
14	tcp	141.192.37.3	*	161.121.34.3	80	DENY
15	tcp	141.192.37.5	*	161.121.34.3	80	DENY
16	tcp	141.192.37.*	*	161.121.34.3	80	ACCEPT
17	tcp	141.193.38.*	*	161.121.34.3	21	ACCEPT
18	tcp	141.193.39.*	*	161.121.34.3	21	ACCEPT
19	tcp	141.192.*.*	*	161.121.34.4	21	ACCEPT
20	tcp	141.*.*.*	*	161.121.34.5	25	ACCEPT
21	udp	142.192.*.*	*	161.122.33.43	69	ACCEPT
22	udp	143.192.*.*	*	161.122.33.43	69	DENY
23	udp	144.*.*.*	*	161.122.33.43	69	ACCEPT
24	udp	145.*.*.*	*	161.122.33.43	69	ACCEPT

Figure 1: A single firewall policy.

protocol a rectangle belongs to. Additional notations are used to clearly indicate those properties. Round brackets are used to tell if a rectangle represents denied traffic, otherwise it represents allowed traffic. Curly brackets are used to denote UDP protocol, otherwise it is TCP protocol.

PolicyVis uses three different rule fields to build the policy graph, two of which are used as the graph's vertical and horizontal coordinates and the third field is integrated into the visualization objects (e.g., at the corner of rectangles) avoiding 3D graphs for simplicity. In general, by default, PolicyVis chooses the investigated scope as one of the coordinates (axes), and from three remaining fields, the least common field will be the other coordinate and the second least common field will be the last dimension.

Besides, PolicyVis places rule field values along x-axis and y-axis in such a way that the aggregated values (e.g., wildcards) precedes the discrete values in the axis, or closer to the origin of the graph. Moreover, the width, the length and the position of a rectangle are chosen based on its corresponding rule's attributes so that an aggregated rule or segment (represented by a rectangle) contains its subset ones in the graph and disjoint segments or rules are represented by non-overlapping rectangles (there are no adjacent rectangles).

Case Studies

In this section, we created application scenarios to explore the potential of PolicyVis to help users find the policy misconfiguration problems. All the scenarios were created based on the single firewall policy shown in Figure 1.

Scenario 1: The admin receives an email from the SSH server development team mentioning that there currently exists a SSH server zero-day exploit in the wild. He wants to investigate the firewall policy for accepted traffic to port 22. The admin performs this investigation by choosing the target (scope): *Destination Port* with 22 as the input and viewing allowed traffic only as shown in Figure 2.

Observation: policy segments that allow traffic to SSH (port 22) are extracted and visualized by PolicyVis as shown in Figure 2. Thus, the admin can then decide to block this traffic temporarily.

Scenario 2: The University's student database is stolen and the database server with IP address 161.120.33.44 (possibly compromised) is suspected not protected well by the firewall. The admin wants to investigate the firewall policy applied to this server. He looks into the traffic allowed and blocked by the firewall for this IP address by choosing the target (scope): *Destination Address* with 161.120.33.44 as the input as shown in Figure 3.

Observation: denied and allowed traffic to port 1433 (MSSQL server) controlled by the firewall is almost like what the admin expected except the traffic from source address 140.192.37.2 (from rule number 1) which should not be allowed. The problem is traffic allowed to 161.120.33.* from rule 1 is also allowed to 161.120.33.44. Thus, the admin might remove or change Rule 1 from the firewall.

Scenario 3: The University's whole network is down because of a denial of service attack. The admin suspects that this attack is from a specific region in a

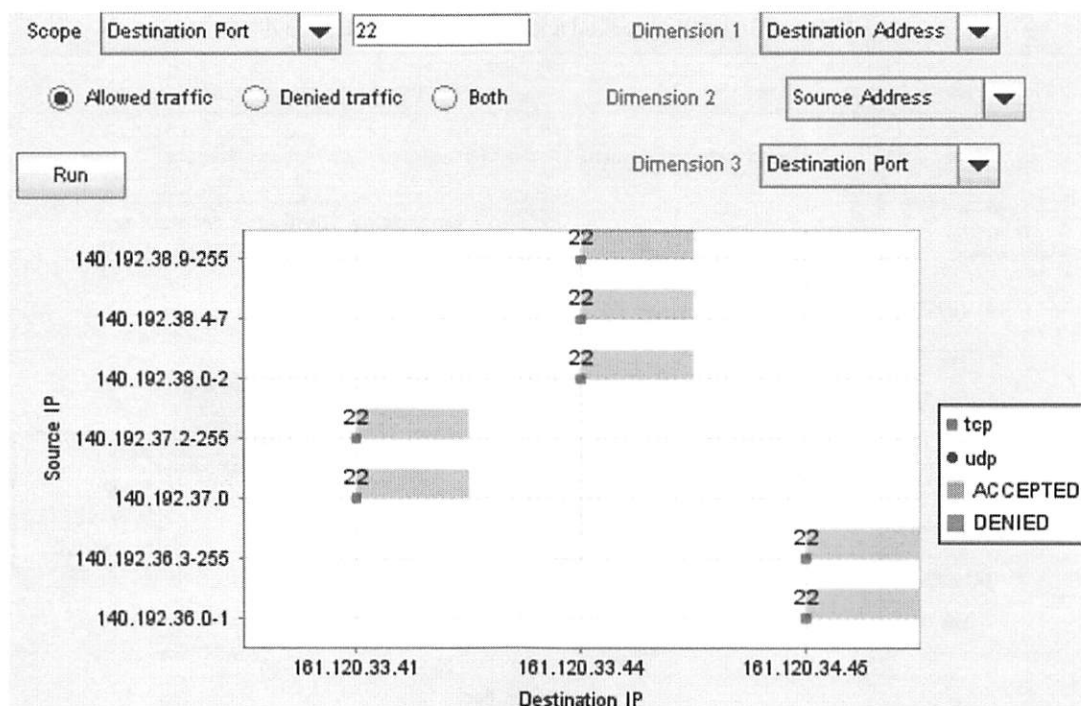


Figure 2: Allowed traffic to port 22.

country with network IP address starting with 141.*.* aiming at many services including telnet, web, ftp, etc. He needs to revise the firewall policy for any traffic from any IP address starting with 141.*.*. The admin chooses the target (scope): *Source Address* with 141.*.* as the input and selects *Destination Port* (corresponding to University's network services) as one of the graph dimensions as shown in Figure 4.

Observation: the firewall policy currently blocks traffic to telnet service (port 23) and web service (port

80) from some IP addresses starting with 141, however, SMTP service (port 25) and FTP service (port 21) are accessible from most of IP addresses starting with 141 and hence vulnerable to the attack. Thus, the admin may set firewall rules to block traffic from some or all addresses starting with 141 to those services as well.

Scenario 4: The University maintains two replicated TFTP servers (port 69) with IP addresses 161.122.33.43 and 161.122.33.44 to satisfy students' high demand of downloading video lectures and also increase

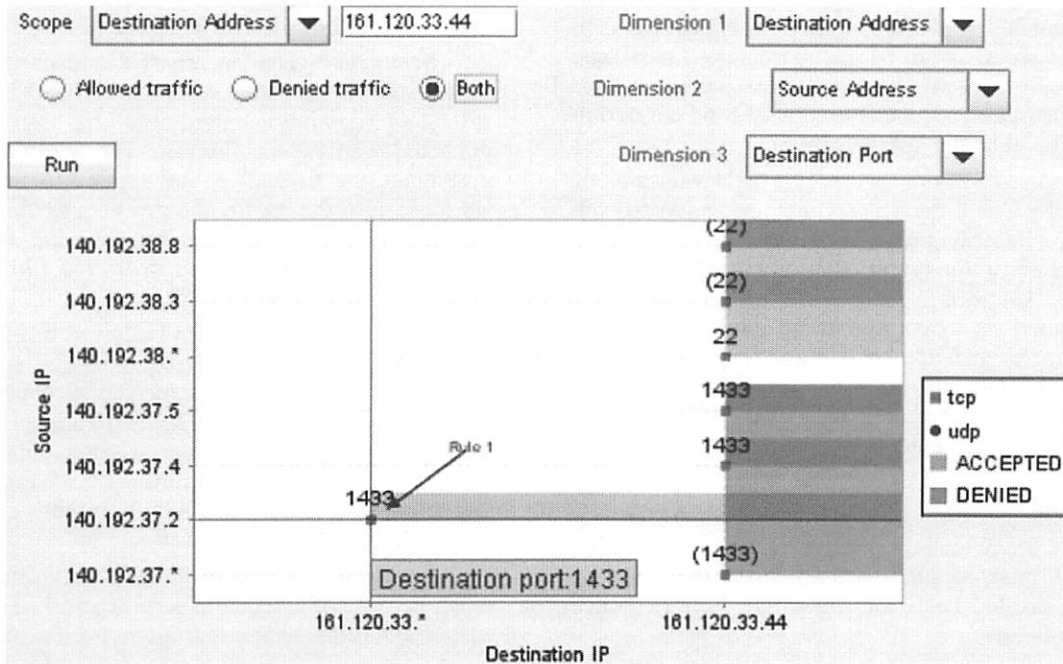


Figure 3: Traffic blocked and allowed to 161.120.33.44.

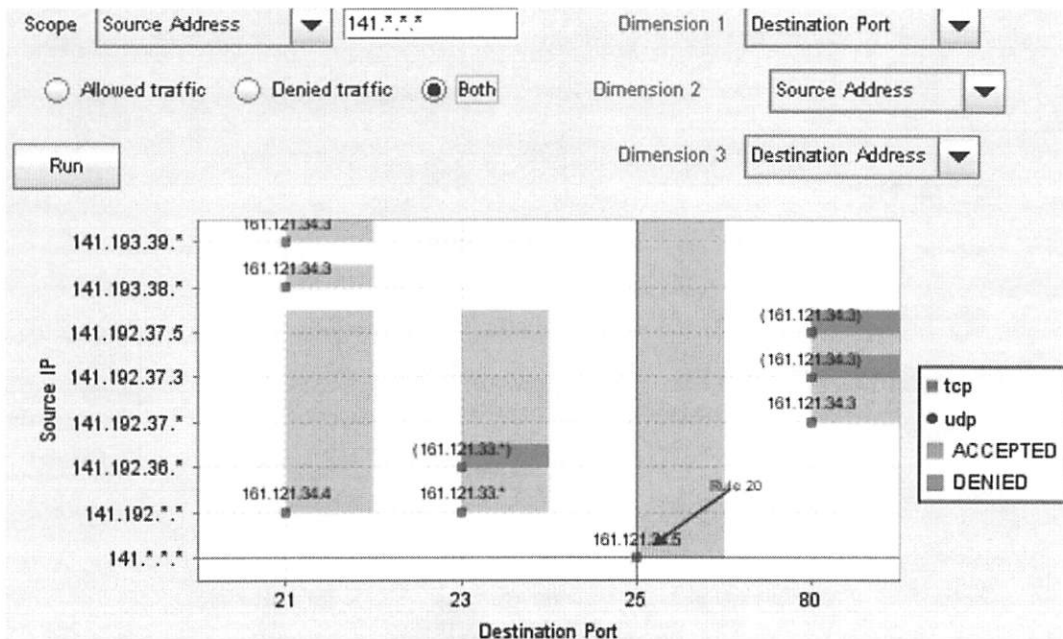


Figure 4: Controlled traffic from 141.*.*.*

the downloading speed. However, several students still complain about low downloading speed and sometimes they are blocked from downloading. The admin first checks the two servers and sees that they both are working well. He suspects that he might make mistakes when writing firewall rules for the two servers so that one of them might not function as wanted. He needs to check the firewall policy and expects that the policy for both servers should be the same because they are replicated and have the same mission. The admin chooses the target (scope): *Destination Port* with 69 as the input as shown in Figure 5.

Observation: traffic controlled by the firewall to the two servers is not the same. The admin recognizes that he made mistakes blocking traffic from 144.*.*.* and 145.*.*.* to server 161.122.33.44 when they should be allowed as to server 161.122.33.43. Thus, the admin corrects his mistakes by changing the actions in the corresponding rules in the firewall.

Visualizing Rule Anomalies

Definition

In this section, we mention crucial definitions and concepts of firewall policy anomalies introduced in [1] so that readers can understand how PolicyVis visualizes rule anomalies described in the next section.

A firewall policy conflict is defined as the existence of two or more filtering rules that may match the same packet or the existence of a rule that can never match any packet on the network paths that cross the firewall, e.g.:

- **Shadowing anomaly:** a rule is shadowed when a previous rule matches all the packets that

match this rule and the shadowed rule will never be activated.

- **Generalization anomaly:** a rule is a generalization of a preceding rule if they have different actions and if the second rule can match all the packets that match the first rule.
- **Redundancy anomaly:** a redundant rule performs the same action on the same packets as another rule such that if the redundant rule is removed, the security policy will not be affected.
- **Correlation anomaly:** two rules are correlated if they have different filtering actions, and the first rule matches some packets that match the second rule and the second rule matches some packets that match the first rule.

Rule Anomaly Visualization Methodology and Algorithm

As the number of firewall rules increases, it is very likely that an anomaly will exist in the policy which threatens the firewall's security. Anomaly discovery is necessary in order to ensure the firewall's concreteness. Firewall policy advisor [1] is the first tool to discover anomalies in a firewall policy. However, it is not as expressive as PolicyVis in anomaly discovery and doesn't give users a visual view on how an anomaly occurs.

Four classes of firewall policy anomalies mentioned previously are visualized by PolicyVis. These anomalies are easily pinpointed by overlapping areas on the graph because an overlapping area represents for rules with overlapping traffic, which can potentially cause firewall policy anomalies. Each of the anomalies has specific features that are easily recognized on the

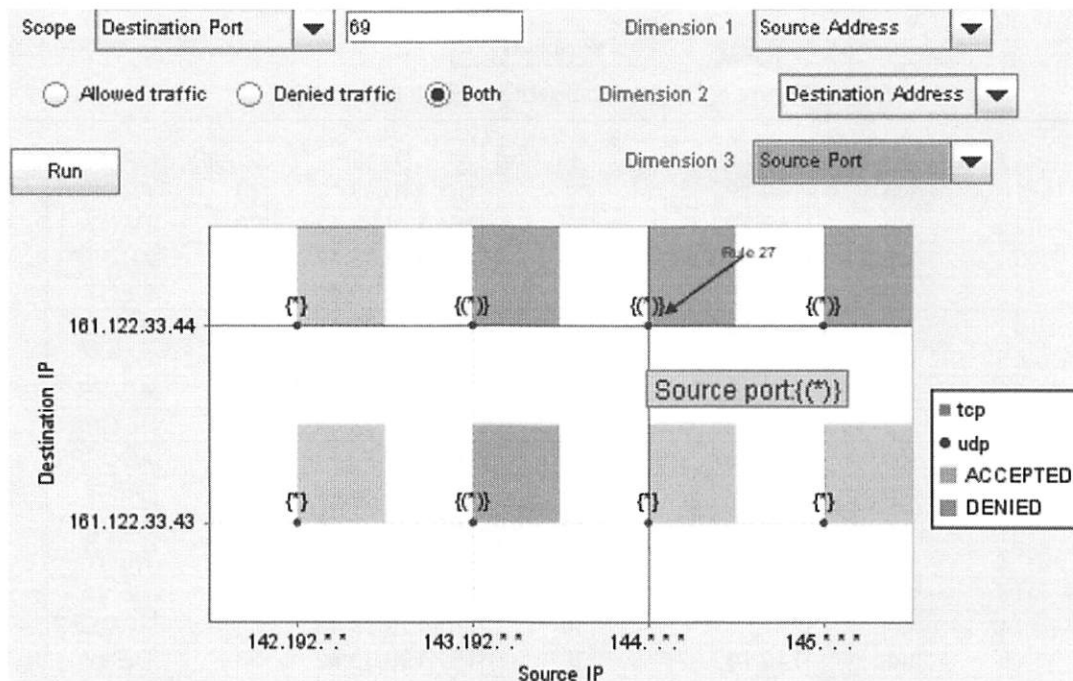


Figure 5: Firewall policy for destination port 69.

PolicyVis graph because its corresponding overlapping area is formed (or look) differently in terms of rectangles position, colors and notations. These features are different for all four anomalies.

As PolicyVis visualizes rules in 2D-graph which shows users only three fields on the graph, an overlapping traffic area is a feature of a potential anomaly, however, it sometimes does not indicate that the corresponding rules are really overlapping because their fourth field might be different. Nonetheless, PolicyVis still lets users visualize real anomalies by allowing related rules to be investigated more closely. When the user wants to investigate an overlapping area, he simply clicks on it and PolicyVis will focus on more details of the related rules.

PolicyVis first collects all rules containing the selected area, and then sketches a different graph for

these rules. In order to correctly view real anomalies with only three fields used on the graph, PolicyVis needs to choose a left-out field which is the same for all the related rules. This common field is guaranteed to exist because related rules from an overlapping area must have at least two fields in common. PolicyVis selects the most common and least important field to be the left-out one if there are multiple common fields among the related rules.

Moreover, among three fields used for the focusing graph, PolicyVis picks the most common field over the related rules to be the third coordinate (the one integrated into visualization objects), and chooses the other two fields as the graph normal coordinates (used for axes). This coordinate selection technique assures users that, from this focusing view, an overlapping area definitely indicates at least one anomaly in the policy.

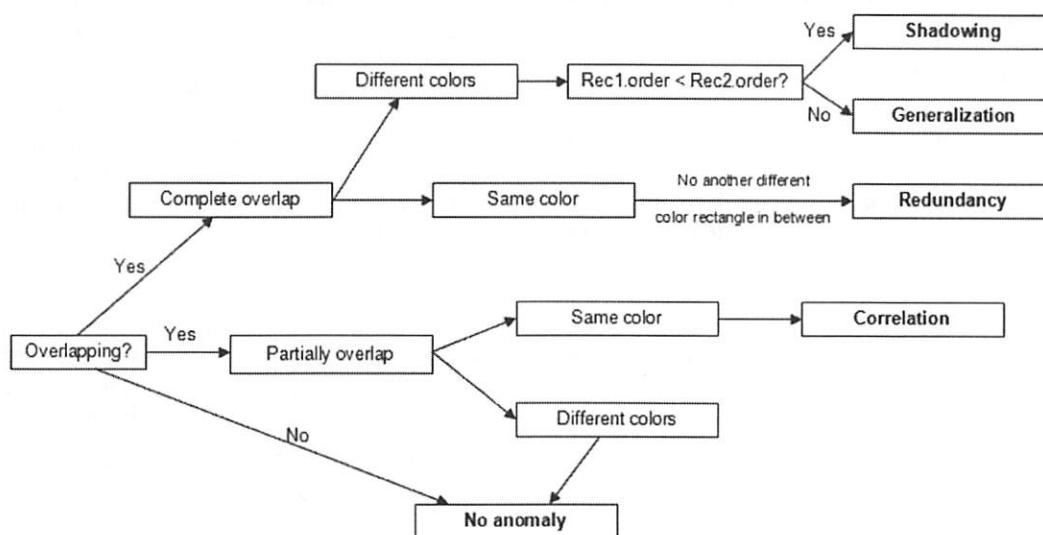


Figure 6: Diagram to determine possible anomalies.

Order	Protocol	SrcIP	SrcPort	DestIP	DestPort	Action
1	tcp	140.192.37.2	*	161.120.33.*	20	ACCEPT
2	tcp	140.192.37.*	*	161.120.33.42	20	DENY
3	tcp	140.192.37.*	*	161.120.33.41	25	ACCEPT
4	tcp	140.192.37.1	*	161.120.33.41	25	DENY
5	tcp	140.192.37.3	*	161.120.33.43	21	ACCEPT
6	tcp	140.192.37.*	*	161.120.33.43	21	DENY
7	tcp	140.192.37.*	*	161.120.33.44	53	ACCEPT
8	tcp	140.192.37.4	*	161.120.33.44	53	ACCEPT
9	tcp	140.192.37.5	*	161.120.33.44	23	ACCEPT
10	tcp	140.192.37.5	*	161.120.33.44	23	DENY
11	tcp	140.192.37.5	34	161.120.33.45	22	ACCEPT
12	tcp	140.192.37.*	35	161.120.33.45	22	DENY
13	tcp	140.192.37.1	*	161.120.33.42	20	DENY
14	udp	***	30	161.120.33.43	50	ACCEPT
15	udp	140.192.37.*	30	161.120.33.43	50	DENY

Figure 7: An example of a firewall policy.

To find the most common field over some firewall rules, for each rule field excluding the Action field, PolicyVis needs to find a rule's field value which is a subset of all other rules' field values, and compute the number of rules that have the field value equal to that rule's field value. The field that has the biggest number is the most common field over the rules. The algorithm FindMostCommonField to find the most common field is implemented as shown in Table 2.

How To Recognize Anomalies In PolicyVis

The rules order is an important factor in understanding the policy semantic and determining the firewall anomaly types, especially between shadowing and generalization anomalies. Besides allowing users to see the rules order by moving the mouse over the overlapping area, PolicyVis also uses surrounding rectangles (not color-filled) around the overlapping rule rectangles only in the focusing view to visualize rules or rectangles order in each overlapping area. The width (and height) difference between a rule rectangle and its surrounding one in an overlapping area is called boarder and it basically shows the rule order: the rule or rectangle with bigger boarder comes first in the policy. This technique will offer an easy way to

determine the type of the anomaly visually and without any manual investigation.

Shadowing and generalization anomalies: These two anomalies can be recognized by a rectangle totally contained in another rectangle but have different colors (different filtering actions), the rules order (based on extra rectangles) will decide which anomaly the overlapping area belongs to.

Redundancy anomaly: The features used to recognize this anomaly are almost the same as features used to pinpoint shadowing and generalization anomalies. Instead of having different colors, the overlapping rectangles should have the same color (same filtering action) and there is no another different color rectangle appears between them.

Correlation anomaly: This anomaly is corresponding to two rectangles with different colors partially contained in each other.

If two rectangles are not overlapping, there is no anomaly between two rules represented by those two rectangles. With the help of PolicyVis, it is straightforward to pinpoint all anomalies that might exist in the firewall policy. Figure 6 summarizes the method to

Algorithm FindMostCommonField

Input: rules

Output: the most common field among the input rules

```

1:  for each field in rule.fields/{action}
2:    if field = dest_ip or field = src_ip
3:      Cfield = *.*.*.*
4:    end if
5:    if field = dest_port or field = src_port
6:      Cfield = *
7:    end if
8:    for each rule Ri in rules { find a field value which is a subset of all other field values }
9:      Cfield = Cfield ∩ Ri.field
10:   end for
11:   Nfield = 0
12:   for each rule i in rules { count the number of rules having field value equal to the common subset value }
13:     if Ri.field = Cfield
14:       Nfield = Nfield + 1
15:     end if
16:   end for
17: end for
18: N = max(Ndest_ip, Nsrc_ip, Ndest_port, Nsrc_port) { choose the most common field }
19: if N = Nsrc_port
20:   return src_port
21: end if
22: if N = Ndest_port
23:   return dest_port
24: end if
25: if N = Nsrc_ip
26:   return src_ip
27: end if
28: if N = Ndest_ip
29:   return dest_ip
30: end if

```

Table 2: Algorithm to find the most common field.

determine different rule anomalies which is very effective in a visualized environment like PolicyVis.

A Case Study

Using PolicyVis to investigate the firewall policy shown in Figure 7, the firewall rules are visualized as shown in Figure 8. The admin sees many overlapping areas which might contain potential rule anomalies. There are five suspected overlapping areas (numbered on the graph) which the user believes contain rule anomalies. From this view only, he suspects that:

1. potential of shadowing anomaly
2. potential of generalization anomaly
3. potential of correlation anomaly

4. potential of redundancy anomaly
5. potential of generalization anomaly

However, in order to make sure that those anomalies are real anomalies, the admin needs to closely investigate each overlapping area. To do this, the admin simply clicks on each selected overlapping area and PolicyVis will focus on and show a more elaborated view for that area.

Shadowing anomaly visualization: When the admin clicks on the overlapping area number 1 (Figure 8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.41* as shown in Figure 9. From this view, it is clear that there

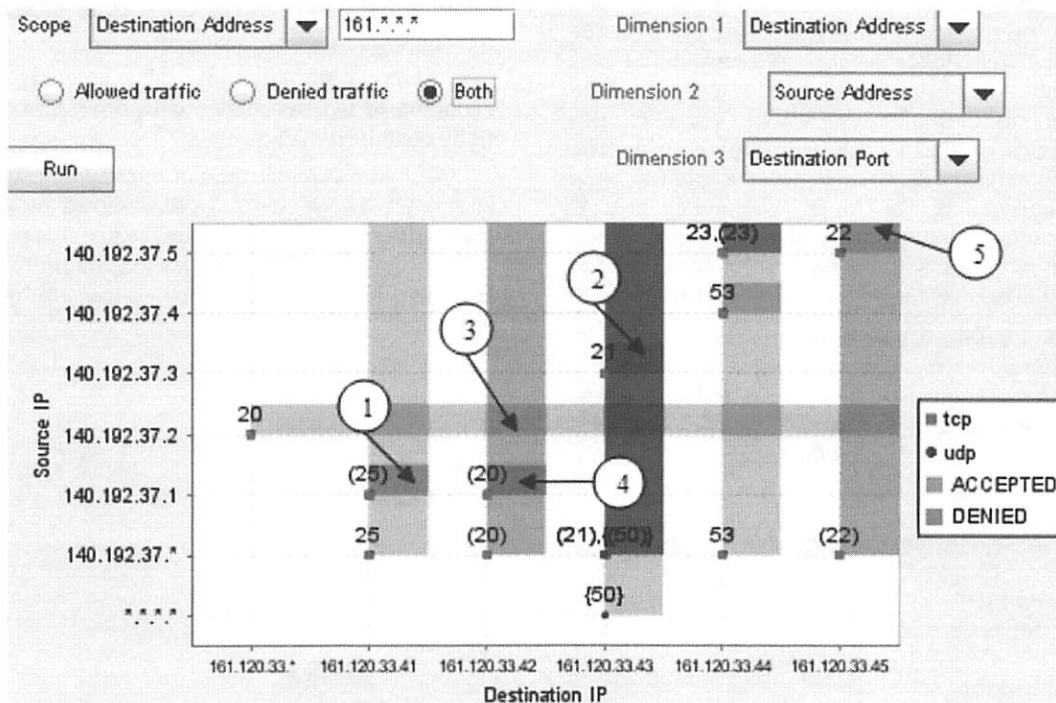


Figure 8: Many potential anomalies in the policy.

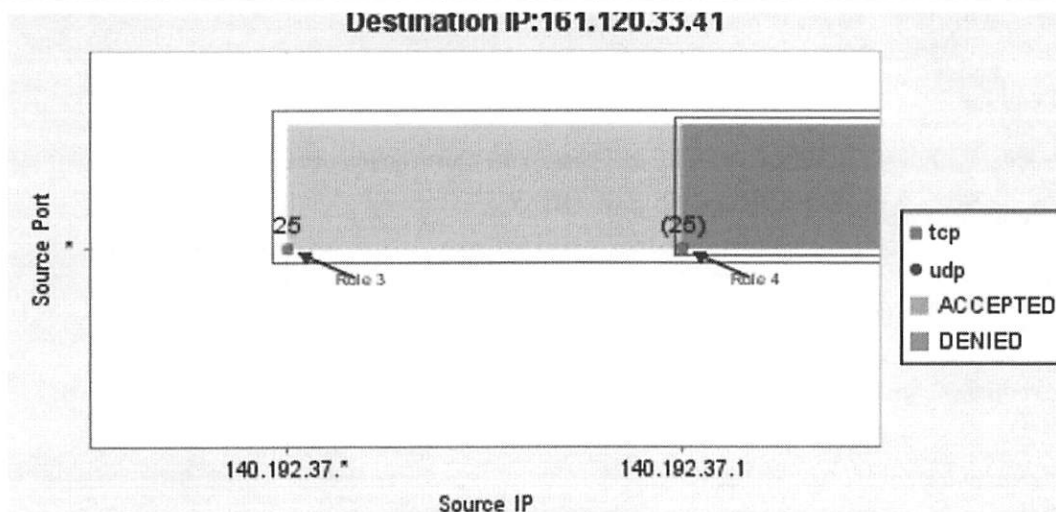


Figure 9: Shadowing anomaly between rule 3 and rule 4.

is a shadowing anomaly between rule 3 and rule 4 (rule 4 is shadowed by rule 3) because the rectangle representing rule 4 is totally contained in the rectangle representing rule 3 and they have different colors. “Rule 3” and “Rule 4” tooltips appear in this case because the admin moves the mouse over the overlapping area. Without these tooltips, the admin still can tell that this is a shadowing anomaly because he knows the outer rectangle comes first in the policy based on the surrounding rectangles.

Generalization anomaly visualization: When the admin clicks on the overlapping area number 2 (Figure 8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.43* as shown in Figure 10. From this view, it is clear that there is a generalization anomaly between rule 5 and rule 6 (rule 6 is a generalization of rule 5) because the rectangle representing rule 5 is totally contained in the rectangle representing rule 6 and they have different colors. Moreover, without the tooltips (“Rule 5” and “Rule 6”), the admin still can tell that the inner rectangle comes first in the policy based on the surrounding rectangles and hence this is a generalization anomaly.

Correlation anomaly visualization: When the admin clicks on the overlapping area number 3 (Figure 8), he is brought to the view where all traffic has the same Destination Port 20 as shown in Figure 11. From this view, it is clear that there is a correlation anomaly between rule 1 and rule 2 because the rectangle representing rule 1 is partially overlapped with the rectangle representing rule 2 and they have different colors.

Redundancy anomaly visualization: When the admin clicks on the overlapping area number 4 (Figure 8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.43* as shown in Figure 12. From this view, it is clear that there is a redundancy anomaly between rule 2 and rule 13 (rule 13 is redundant to rule 2) because the rectangle representing rule 13 is totally contained in the rectangle representing rule 2 and they have the same color.

Overlap but no anomaly: When the admin clicks on the overlapping area number 5 (Figure 8), he is brought to the view where all traffic has the same Destination IP address *161.120.33.45* as shown in Figure 13. From this view, it is clear that there is no anomaly because the rectangles representing rules are

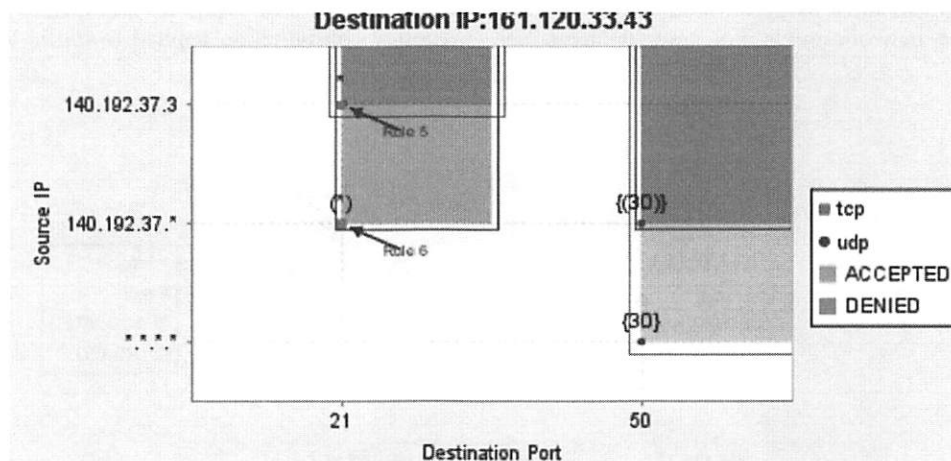


Figure 10: Generalization anomaly between rule 5 and rule 6.

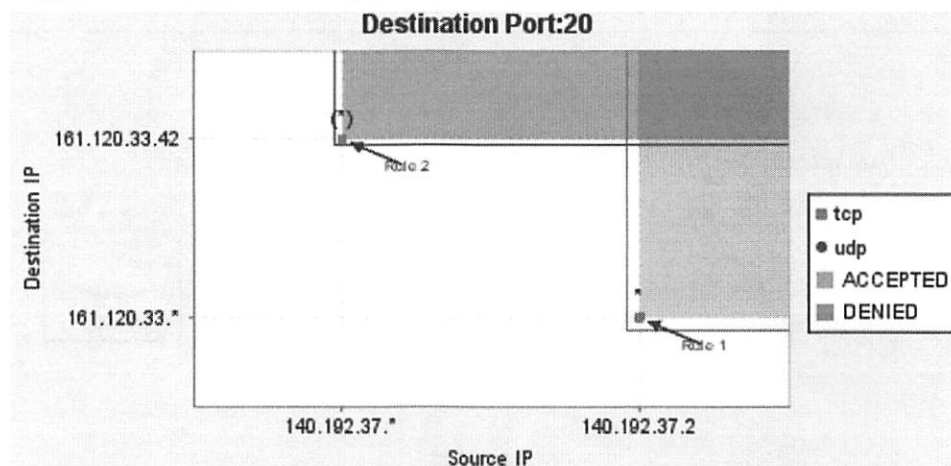


Figure 11: Correlation anomaly between rule 1 and rule 2.

not overlapping. Rule 11 and Rule 12 are overlapped in Figure 8 because Rule 11's Destination Address and Source Address are subsets of Rule 12's Destination Address and Source Address respectively and those two fields with Destination Port are chosen as dimensions for the view as shown in Figure 8. However, Rule 11 and Rule 12 have different Source Ports which is automatically chosen by PolicyVis as one of the dimensions for the new view as shown in Figure 13.

Visualizing Distributed Policy Configuration

Concept

While a single firewall is normally deployed to protect a single subnet or domain, distributed firewalls are essential for protecting the entire network. Any misconfiguration or conflict between distributed firewalls might cause serious flaws or damages to the network [2].

Anomalies exist not only in a single firewall but also in inter-firewalls if any two firewalls on a network path take different filtering actions on the same traffic. It is always a higher chance that distributed firewalls contain rule anomalies than a single firewall because of the decentralized property in distributed firewalls management. It is possible that each single firewall in the network might not contain any rule

anomaly, but there are still anomalies between different firewalls.

Visualizing distributed firewalls gives the same benefits as visualizing single firewalls in achieving policy behavior discovery, policy correctness checking and anomaly finding. Distributed firewalls are considered as a tree where the root is the borderline firewall which directly filters traffic in and out of the network. Each node in the tree represents for a single firewall which can be placed between subnets or domains in the network.

A packet from outside of the network in order to get through a firewall needs to pass all filterings of all firewalls from the root to the node representing that firewall. In the distributed firewalls view, PolicyVis creates a firewall tree based on the network topology input files and let the user pick a path (from the root to any node) he wants to examine. PolicyVis then builds up a rule set for that path by simply reading rules from nodes in order from the root to the last node. After that, PolicyVis considers this rule set as for a single firewall and visualizes it as before.

A Case Study

The admin wants to investigate the distributed policy configuration applied to traffic to the *Network*

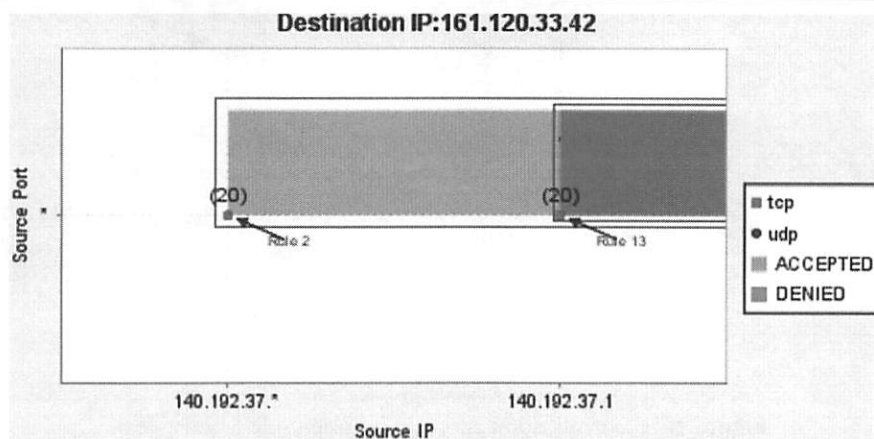


Figure 12: Redundancy anomaly between rule 2 and rule 13.

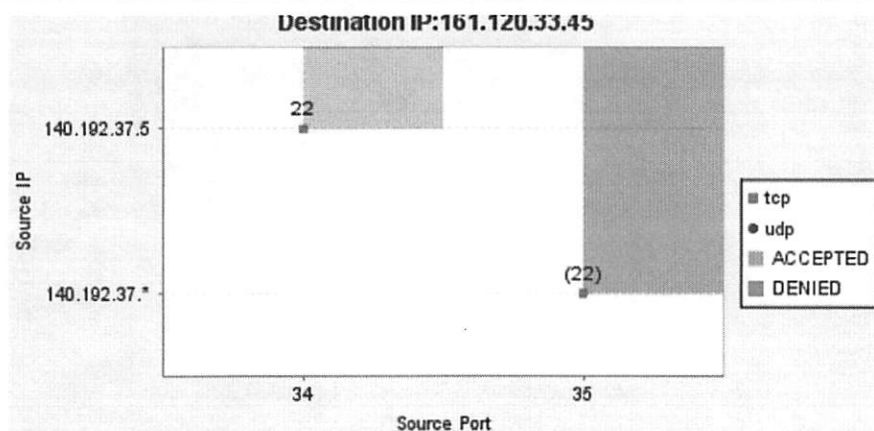


Figure 13: There is no anomaly in this case.

Lab. He first changes the view to *Distributed Firewalls* view and expands the tree to get to the *Network Lab* node. As shown in Figure 14, PolicyVis creates a new rule set containing all rule sets from firewalls on the path in this order: *University of Waterloo*, *Math faculty*, *CS department*, and *Network Lab*.

After building up the rule set for the path from *University of Waterloo* to *Network Lab*, PolicyVis allows the admin to start visualizing the path policy. In this visualization, the admin chooses to investigate all rules on this firewall path that control traffic to any destination address in the university network by choosing the scope *Destination Address* with value *161.*.*.**.

In this case, there are normally multiple subnets get involved because multiple firewalls are considered at once. PolicyVis not only lets the admin visualize all the subnets at the same time, but also supports a single view on each subnet and the admin can switch views between subnets easily. In this example, there are six subnets whose traffic are controlled by the firewalls on the path and the *Network Lab* subnet *161.120.33.** is currently viewed and analyzed by the admin (Figure

15). The admin can change the view to a different subnet by clicking on the *Next* or *Previous* button.

It is easy to recognize that while the single firewall placed at the *Network Lab* subnet (Figure 16) which only controls traffic to *161.120.33.** doesn't contain any anomaly, the distributed firewalls (Figure 15) seems to have anomalies (overlapping areas). In fact, there is a shadowing anomaly in this case between a rule in the *University of Waterloo* firewall and a rule in the *Network Lab* firewall.

Implementation and Evaluation

We implemented PolicyVis using Java and Jfreechart [7], a free open source Java chart library, in PolicyVis to make it easy for displaying charts in the graph. We also used Buddy [17] for BDD representation of firewall policies.

In this section, we present our evaluation study of the usability and efficiency of PolicyVis. To access the practical value of PolicyVis, we not only created firewall policies randomly (with and without rule anomalies), but also used real firewall rules in our

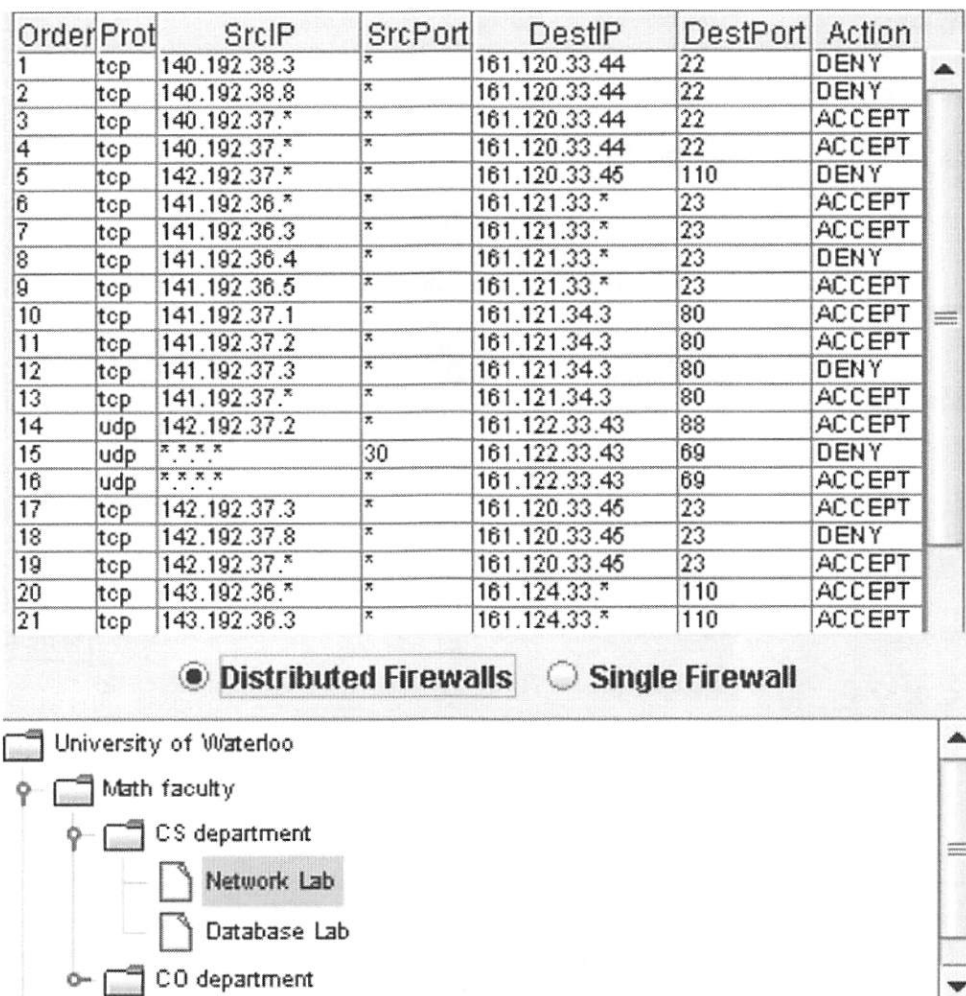


Figure 14: An example of distributed firewalls.

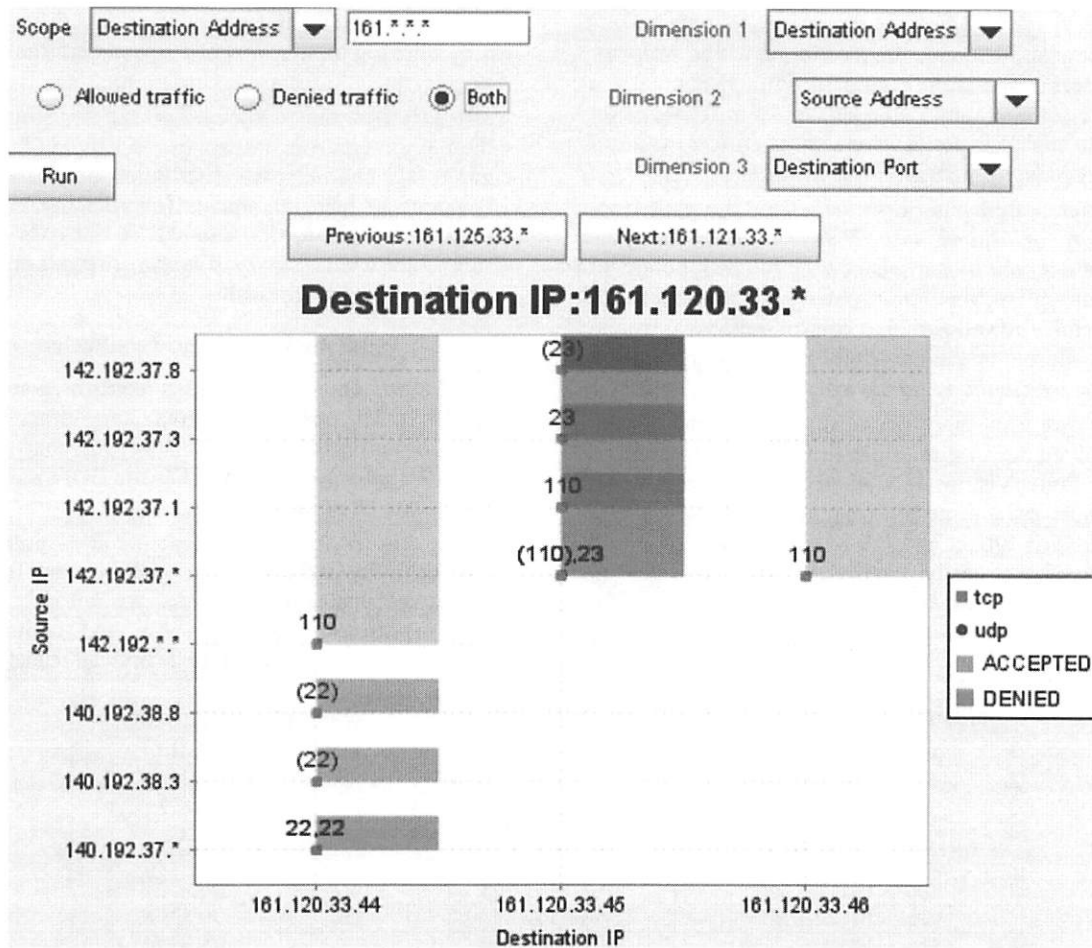
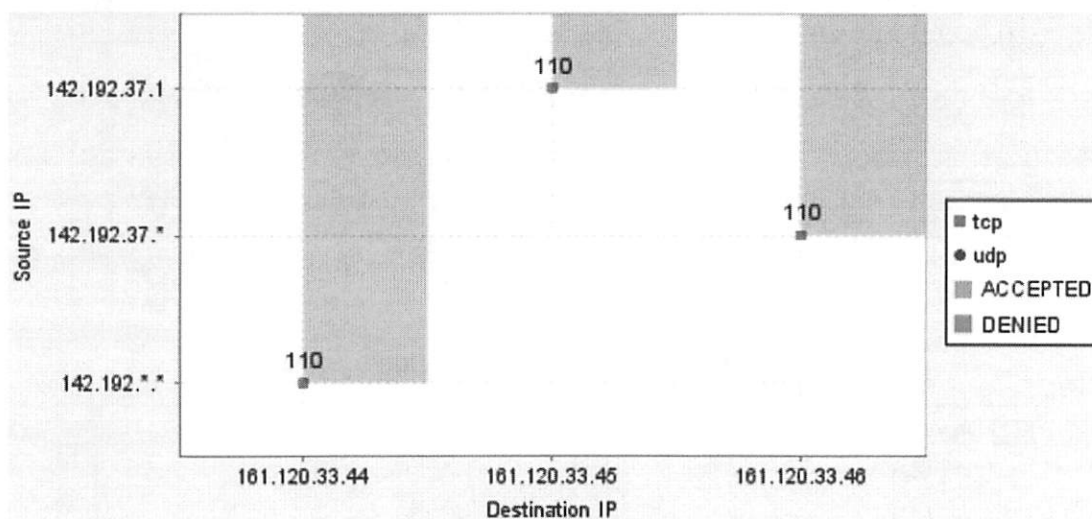


Figure 15: Visualization of all firewall policies to subnet 161.120.33.*.

Figure 16: Visualization of the *Network Lab* subnet firewall.

<i>Task\Method</i>	<i>PolicyVis</i>	<i>Raw firewall rules</i>
Find firewall properties	3.12 minutes	10.44 minutes
Find firewall anomalies	1.98 minutes	12.78 minutes

Table 3: Average estimated time to achieve each task by using each method.

university for the evaluation study. Each firewall used in the evaluation test has from 30 to 45 rules. We then asked 11 people (with varying level of expertise in the field) under test to use both PolicyVis and raw firewall rules to find some specific firewall properties (like what traffic is allowed to a chosen domain or which machine has Web accessible web traffic and so on) and locate rule anomalies in the firewalls. We recorded the time to answer each task by using each method for all people and computed the average time over all.

People in this evaluation test were getting familiar with PolicyVis very quickly and very confident with features supported by PolicyVis. As shown in Table 3, the average time to achieve each task by using PolicyVis is much faster than by investigating raw firewall rules, especially in finding firewall anomalies. This evaluation test demonstrated that PolicyVis is a very user-friendly tool with high usability and efficiency.

Conclusion and Future Work

Firewalls provide proper security services if they are correctly configured and efficiently managed. Firewall policies used in enterprise networks are getting more complex as the number of firewall rules and devices becomes larger. As a result, there is a high demand for an effective policy management tool which significantly helps user in discovering firewall policy's properties and finding rule anomalies in both single and distributed firewalls.

PolicyVis presented in this paper provides visual views on firewall policies and rules which gives users a powerful means for inspecting firewall policies. In this paper, we described design features of PolicyVis tool and illustrated PolicyVis with multiple examples showing the effectiveness and usefulness of PolicyVis in determining the policy behavior in various case studies. We presented concepts and techniques to find rule anomalies in PolicyVis. Besides, we also showed how PolicyVis visualizes distributed firewalls to achieve same benefits as visualizing single firewalls. Finally, we presented the implementation and evaluation of PolicyVis.

Using PolicyVis was shown very effective for firewalls in real-life networks. In regards to usability, unskilled people with short time of learning how to use PolicyVis can quickly understand and start using all features of PolicyVis. Moreover, by evaluation, PolicyVis effectively helped users including network security juniors and seniors to figure out firewall policy behavior easily by reviewing the visualizing of primitive firewall rules. In addition, PolicyVis was shown a very good tool in finding rule anomalies or conflicts easily and quickly. The number of dimensions users need to consider during firewall inspection varies according to situations; however, considering all possible rule fields always gives users a better analysis of the policy.

Even though PolicyVis was shown a very effective tool, we still want to perform more evaluation on it and

collect more users' ideas to make PolicyVis the best visualization tool for firewall policies. There are still many possible features that we want to implement in PolicyVis to maximize its usability as well as efficiency. We want PolicyVis to support more viewing levels of firewall policies and automatically show users possible strange behaviors and true rule anomalies of firewall policies on the graph. In addition, PolicyVis currently shows how to visualize stateless firewalls but we can easily envision extending this to visualize stateful firewalls too by preprocessing the policy to create the implicit rules in stateful firewalls. We consider supporting stateful firewalls in PolicyVis in our future work.

Author Biographies

Tung Tran received the Bachelor of Math with Distinction on the Dean's Honours List from the University of Waterloo in 2006. He is currently a Master student of Computer Science at the University of Waterloo, under the co-supervision of Prof. Raouf Boutaba and Prof. Ehab Al-Shaer. His main research concentrates on network security, especially firewall and IDS policy management. He can be reached at t3tran@cs.uwaterloo.ca.

Ehab Al-Shaer is an Associate Professor and the Director of the Multimedia Networking Research Lab (MNLAB) in the School of Computer Science, Telecommunications and Information System at DePaul University since 1998. Prof. Al-Shaer is also a Research Scientist faculty at School of Computer Science, University of Waterloo.

His primary research areas are Network Security, fault and Configuration management. He is Co-Editor of number of books in the area of multimedia management and end-to-end monitoring. Prof. Al-Shaer has serves as a Program Co-chair for number of well-established conferences in area including IM 2007, POLICY 2008, MMNS 2001, and E2EMON 2003-2005. Prof. Al-Shaer was a Guest Editor for number of Journals in his area. He has also served as TPC member, session chair, and tutorial presenter for many IEEE/ACM/IFIP conferences in his area including INFOCOM, ICNP and IM/NOMS and others. His research is funded by NSF, Cisco, Intel, and Sun Microsystems. Prof. Al-Shaer has received several best paper awards and other awards such as a NASA fellowship. He received his M.S. and Ph.D. in Computer Science from Northeastern University, Boston, MA and Old Dominion University, Norfolk, VA in 1994 and 1998 respectively.

Raouf Boutaba received the M.Sc. and Ph.D. Degrees in Computer Science from the University Pierre & Marie Curie, Paris, in 1990 and 1994 respectively. He is currently a Professor of Computer Science at the University of Waterloo. His research interests include network, resource and service management in multimedia wired and wireless networks.

Dr. Boutaba is the founder and Editor-in-Chief of the IEEE Transactions on Network and Service Management and on the editorial boards of several other journals. He is currently a distinguished lecturer of the IEEE Communications Society, the chairman of the IEEE Technical Committee on Information Infrastructure and the IFIP Working Group 6.6 on Network and Distributed Systems Management. He has received several best paper awards and other recognitions such as the Premier's research excellence award.

References

- [1] Al-Shaer, E., H. Hamed, and R. Boutaba, M. Hasan, "Conflict Classification and Analysis of Distributed Firewall Policies," *IEEE Journal of Selected Areas of Communications (JSAC)*, 2005.
- [2] Al-Shaer, E. and Hazem Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *Proceedings of IEEE INFOCOM'04*, March, 2004.
- [3] Becker, R. A., S. G. Eick, and A. R. Wilks, "Visualizing Network Data," *IEEE Transactions on Visualization and Computer Graphics*, pp. 16-28, 1995.
- [4] Bethel, E. W., S. Campbell, E. Dart, K. Stockinger, and K. Wu, "Accelerating Network Traffic Analysis Using Query-Driven Visualization," *IEEE Symposium on Visual Analytics Science and Technology*, IEEE Computer Society Press, 2006.
- [5] El-Atawy, A., K. Ibrahim, H. Hamed, and E. Al-Shaer, "Policy Segmentation for Intelligent Firewall Testing," *1st Workshop on Secure Network Protocols (NPsec 2005)*, November, 2005.
- [6] Eppstein, D. and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," *Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January, 2001.
- [7] Gilbert, David and Thomas Morgner, *JFreeChart, Java Chart Library*, <http://www.jfree.org/jfreechart/>.
- [8] Goodall, John, Wayne Lutters, Penny Rheingans, and Anita Komlodi, "Preserving the Big Picture: Visual Network Traffic Analysis with TNV," *Proceedings of the 2005 Workshop on Visualization for Computer Security*, pp. 47-54, October, 2005.
- [9] Hamed, Hazem, Ehab Al-Shaer and Will Marro, "Modeling and Verification of IPSec and VPN Security Policies," *Proceedings of IEEE ICNP'2005*, November, 2005.
- [10] Hari, B., S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," *Proceedings of IEEE INFOCOM'00*, March, 2000.
- [11] Lakkaraju, K., R. Bearavolu, and W. Yurcik, "NVisionIP - A Traffic Visualization Tool for Large and Complex Network Systems," *International Multiconference on Measurement, Modeling, and Evaluation of Computer-Communication Systems (Performance TOOLS)*, 2003.
- [12] Lee, Chris P., Jason Trost, Nicholas Gibbs, Raheem Beyah, John A. Copeland, "Visual Firewall: Real-time Network Security Monitor," *Proceedings of the IEEE Workshops on Visualization for Computer Security*, p. 16, October 26-26, 2005.
- [13] Liu, A. X., M. G. Gouda, H. H. Ma, and A. H. Ngu, "Firewall Queries," *Proceedings of the 8th International Conference on Principles of Distributed Systems*, LNCS 3544, T. Higashino Ed., Springer-Verlag, December, 2004.
- [14] Le Malécot, E., M. Kohara, Y. Hori, and K. Sakurai, "Interactively combining 2D and 3D visualization for network traffic monitoring," *Proceedings of the 3rd ACM international Workshop on Visualization For Computer Security*, November 3, 2006.
- [15] McPherson, J., K. L. Ma, P. Krystosk, T. Bartolletti, and M. Christensen, "PortVis: A Tool for Port-Based Detection of Security Events," *Proceedings of CCS Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC)*, October, 2004.
- [16] Nidhi, Sharma, *FireViz: A Personal Firewall Visualizing Tool*, Thesis (M. Eng.), Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2005.
- [17] Lind-Nielsen, J., *The Buddy obdd Package*, <http://www.bddportal.org/buddy.html>.
- [18] Wool, A., "Architecting the Lumeta Firewall Analyzer," *Proceedings of 10th USENIX Security Symposium*, August, 2001.
- [19] Tufin SecureTrack: Firewall Operations Management Solution, <http://www.tufin.com>.

Inferring Higher Level Policies from Firewall Rules

Alok Tongaonkar, Niranjana Inamdar, and R. Sekar – Stony Brook University

ABSTRACT

Packet filtering firewall is one of the most important mechanisms used by corporations to enforce their security policy. Recent years have seen a lot of research in the area of *firewall management*. Typically, firewalls use a large number of low-level filtering rules which are configured using vendor-specific tools. System administrators start off by writing rules which implement the security policy of the organization. They add/delete/change order of rules as the requirements change. For example, when a new machine is added to the network, new rules might be added to the firewall to enable certain services to/from that machine. Making such changes to the low-level rules is complicated by the fact that the effect of a rule is dependent on its priority (usually determined by the position of the rule in the rule set). As the size and complexity of a rule set increases, it becomes difficult to understand the impact of a rule on the rule set. This makes management of rule sets more error prone. This is a very serious problem as errors in firewall configuration mean that the desired security policy is not enforced.

Previous research in this area has focused on either building tools that generate low-level firewall rules from a given security policy or finding anomalies in the rules, i.e., verifying that the rules implement the given security policy correctly. We propose a technique that aims to infer the high-level security policy from low-level representation. The first step in our approach is that of generating *flattened rules*, i.e., rules without priorities, which are equivalent to the given firewall rule set. Removal of priorities from a rule set enables us to merge a number of rules that have a similar effect. Our rule merging algorithm reduces the size and complexity of the rule set significantly by grouping the *services*, *hosts*, and *protocols* present in these rules into various (possibly overlapping) classes. We have built a prototype implementation¹ of our approach for *iptables* firewall rules. Our preliminary experiments indicate that the technique infers security policy that is at a sufficiently high level of abstraction to make it understandable and debuggable.

Introduction

Firewalls are the first line of defense for protecting corporate networks. System administrators use packet filtering firewalls as one of the mechanisms to implement the security policy of an enterprise. These firewalls are configured using rules that specify matching criteria, and the action to be performed when a packet matches each rule. These rules are matched sequentially against all packets passing through the firewall. These rules can be conflicting, i.e., multiple rules with different actions can match a packet. In such a case, the priority of the rules in the rule set determines the action to be performed. Typically, firewalls use a *first match* policy, i.e., the action corresponding to the first matched rule is taken irrespective of the other rules that can match the packet. Thus the order of rules in a firewall rule set defines a priority relation over the rules. Understanding the effect of firewall rules on network traffic is complicated by this priority relation between the rules.

System administrators initially configure the firewalls with rules that implement the security policy of the organization. As the requirements of the enterprise

change, new rules are added or deleted from the rule set without *refactoring*. Over time, the rule set contains many rules which are very similar and the mapping between the security policy and the rules becomes unclear. Managing such large rule sets becomes increasingly difficult leading to configuration errors which are a serious security concern [10]. Hence, firewall management tools become necessary to help system administrators.

Many tools for firewall management (e.g., Firmato [2], Firestarter [3], Shorewall [4]) focus on generating low-level rules from high-level policy language (or GUI). Recent years have seen many works [6, 13, 1] which try to discover configuration errors in the firewalls. But tools which aid in understanding existing firewall rule sets are missing from the arsenal of system administrators. Some tools (e.g., ITVal [8, 9], Fang [7]) provide a way of *querying* whether certain packets will be allowed through the firewall.

The problem with such tools is that the administrator has to know what to query for. Tools like Lumeta Firewall Analyzer [12] try to avoid this problem by automating the task of querying the firewalls.

¹This research is supported by NSF grants 0208877 and 0627687. Lumeta Firewall Analyzer queries the firewall for all

possible packets that are allowed to pass. For medium to large rule sets this results in a large amount of data being generated. Analyzing such large amounts of data presents another challenge to the system administrator.

We present a novel way to address this problem in this paper. Our approach of inferring the high-level policy from low-level packet filtering rules presents the information to the user in a compact format. Figure 1 shows 24 *iptables* rules taken from a larger firewall rule set (65 rules) being used for a network within our department.² It is quite difficult to understand what kind of traffic is allowed through the firewall looking at the script. A new system administrator who is assigned to manage a firewall rule set like this needs to understand the security policy so that she may answer questions such as:

- which services are allowed on each host?
- which hosts are allowed to communicate with each other?
- what protocol is valid between communicating hosts?

Figure 2 shows the 10 rule policy generated by our technique for the same rule set.³ Clearly it is easier for a system administrator to understand this policy than the rule set in the Figure 1. Moreover, the high-level policy can reveal opportunities for refactoring the low-level rules.

System administrators have an intuitive notion of whether a policy is “complicated” or “simple.” The complexity of a policy depends not just on the number of rules in the policy but also on how complicated those rules are. In this work, we define a metric for the

²We have modified the IP addresses due to privacy concerns.

³Rules in flattened rule set and high-level policy are labeled with alphabets to emphasize the fact that these rules do not have any priority relation defined over them

complexity of a rule set/policy that captures this notion. This allows us to compare different representations of the same rule sets. Our technique infers policies with low complexity and hence these policies are easier to understand.

Our objective was to develop a technique to infer firewall policies that would help the system administrators to work at a higher level of abstraction. Our technique can be combined with existing techniques to form a comprehensive firewall management toolkit. The benefits of such a toolkit are clear from the following scenario: a system administrator who needs to modify some existing legacy firewall rule set can extract the security policy from the rule set using our technique. She can then make changes to the high-level policy and use an automated tool to generate the low-level rules.

Since our technique uses decision tree like graphs (explained in Section Priority Elimination Phase) to represent the firewall rules, it is easy to enhance our system to provide *querying* facility. Moreover, our system automatically removes redundant rules from the policy. Hence, it is a trivial task to identify such redundant rules in the input rule set using our technique.

We initially present an overview of our approach. The next two sections provide the details of the components in our system. We then discuss related work followed by concluding remarks in final section.

Approach Overview

Our approach for inferring policy consists of two phases. First, in *priority elimination* phase, we convert the low-level rule set that contains rules with priorities to an equivalent rule set that contains rules with no priority relation defined over them. We call the generated rules as *flattened rules*. The flattened rule set

```

1. IPTABLES -A FORWARD -p tcp -d 192.168.1.250 --dport domain -j ACCEPT
2. IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport smtp -j ACCEPT
3. IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport smtps -j ACCEPT
4. IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport imaps -j ACCEPT
5. IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport pop3s -j ACCEPT
6. IPTABLES -A FORWARD -p tcp -d 192.168.1.252 --dport www -j ACCEPT
7. IPTABLES -A FORWARD -p tcp -d 192.168.1.126/25 --dport auth -j ACCEPT
8. IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.13 --dport ssh -j ACCEPT
9. IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.14 --dport ssh -j ACCEPT
10. IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.15 --dport ssh -j ACCEPT
11. IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.20 --dport ssh -j ACCEPT
12. IPTABLES -A FORWARD -p tcp -d 192.168.1.252 --dport https -j ACCEPT
13. IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p tcp --dport sunrpc -j ACCEPT
14. IPTABLES -A FORWARD -s 192.168.1.236 -p tcp -d 192.168.1.35 --dport ipp -j ACCEPT
15. IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport nfs -j ACCEPT
16. IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport 4000:4002 -j ACCEPT
17. IPTABLES -A FORWARD -p udp -d 192.168.1.251 --dport smtp -j ACCEPT
18. IPTABLES -A FORWARD -p udp -d 192.168.1.250 --dport domain -j ACCEPT
19. IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport sunrpc -j ACCEPT
20. IPTABLES -A FORWARD -s 192.168.1.236 -p udp -d 192.168.1.35 --dport ipp -j ACCEPT
21. IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type destination-unreachable -j ACCEPT
22. IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type parameter-problem -j ACCEPT
23. IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type source-quench -j ACCEPT
24. IPTABLES -A FORWARD -j REJECT

```

Figure 1: Sample *iptables* script.

does not contain any overlapping rules, i.e., there is one and only one flattened rule that can match a given packet. This simplifies the process of inferring policies from rule sets. Unlike the original rules, flattened rules can be arbitrarily reordered without modifying their overall effect. This enables us to reorder and merge similar rules together, thereby reducing the size and complexity of the generated policy.

The problem with priority elimination is that it generates a large number of rules. In the *policy inference* phase, we reduce the number of rules by grouping hosts, services, and protocols into (possibly overlapping) classes and merging rules containing same class of objects. It is not sufficient to produce rule sets with small number of rules as the complexity of the generated rules also affects the complexity of the entire rule set. Arbitrary merging of rules can lead to rule sets which are very complicated. So this phase tries to merge the rules such that the complexity of the inferred policy is minimized. Finally, the inferred policy is presented to the user.

Background

For concreteness, we describe the details of *iptables*. Almost every packet filtering firewall relies on the type of rules used in *iptables*. *Iptables* [14] is the user space command line program used to configure the rule set in the *netfilter* framework in Linux 2.4.x and 2.6.x. *Netfilter* framework enables packet filtering, network address (and port) translation (NA[P]T) and other packet mangling. *Iptables* can be used to configure three independent tables – *filter*, *nat*, and *mangle*, within the kernel. The *filter* table is used to set up rules that are used for filtering packets, while the *nat* table is consulted when a packet that creates a new connection is encountered and the *mangle* for specialized packet alteration.

In this paper, we are concerned only with the *filter* table which is used as a packet filtering firewall. The *filter* table consists of ordered lists of rules that are called *chains*. The order of rules in a chain determines their priority. There are three built-in chains in the *filter* table. INPUT chain is used to filter packets that are destined for the host on which the firewall is running. OUTPUT chain is used to filter packets generated by the firewall host. FORWARD chain is used to filter packets forwarded by the firewall host to other hosts in the network. One powerful feature of *iptables*

is that it allows the user to define new chains in addition to the built-in ones. This allows the administrators to group rules which together provide certain high-level function like protecting a subnet.

An *iptables* rule consists of matching criteria and the target. Target specifies the action to be taken when a packet satisfies the matching criterion. Matching criteria is specified in terms of tests on the packet header fields like destination IP address (dhost), source IP address (shost), destination port (dport), source port (sport), protocol (proto). Target can be any of the following: ACCEPT, QUEUE, REJECT, DROP, LOG or name of a user defined chain.

Target ACCEPT means that the packet is to be allowed to pass through the firewall. QUEUE passes the packets on to the user space. For our purposes, semantically QUEUE is similar to ACCEPT as the packet is allowed to reach its destination. So we treat QUEUE just like ACCEPT and omit it from our discussion. DROP and REJECT mean the packet is to be denied. REJECT returns an icmp error packet to the sender while DROP denies the packet without giving any error indication. Target LOG on the other hand just makes an entry to the log file when a matching packet arrives. By specifying user defined chains as targets, conditional call/return semantics can be added to the firewall rule set.

Figure 3 shows a sample *iptables* rule set. All the rules considered are for the FORWARD chain with a default policy of REJECT. Rule 1 specifies that all hosts from the network 192.168.1.0/24 are allowed to connect to the host 120.240.18.1 using SMTP. Rule 2 specifies that host 120.240.18.1 can connect to the network 120.240.20.0/24 using SMTP. This can be a real world scenario where 192.168.1.0/24 is an internal network of an organization, 120.240.20.0/24 is external network and 120.240.18.1 is the SMTP server for that organization. The rule set says that SMTP server can send SMTP traffic to external network and internal hosts can send SMTP traffic to SMTP server but not to the external network.

We represent the *iptables* rules in tabular format for ease of understanding. Table 1 is the tabular representation of the rules in Figure 3. We list all rules in a table in the order of their priority. The columns indicate the packet fields being tested. A rule is represented as a row with values for the packet fields being

Allow only the following packets:

- a. tcp, udp FROM 192.168.1.254/28 TO 192.168.1.11 FOR sunrpc
- b. udp FROM 192.168.1.254/28 TO 192.168.1.11 FOR nfs, ports [4000-4002]
- c. tcp FROM 192.168.1.126/25 TO [192.168.1.13 - 15], 192.168.1.20 FOR ssh
- d. tcp, udp FROM 192.168.1.236 TO 192.168.1.35 FOR ipp
- e. tcp TO 192.168.1.126/25 FOR auth
- f. icmp TO 192.168.1.126/25 OF TYPES destination-unreachable, parameter-problem, source-quench
- g. tcp, udp TO 192.168.1.250 FOR domain
- h. tcp, udp TO 192.168.1.251 FOR smtp
- i. tcp TO 192.168.1.251 FOR smtps, imaps, pop3s
- j. tcp TO 192.168.1.252 FOR www, https

Figure 2: Higher level policy for rules in Figure 1.

tested filled in the respective columns. If a rule does not contain any test on a particular field, then that column has a wild-card character “*”. A “*” for a field indicates that any value of the field will match this rule. The action associated with a rule is shown in the last column. Note that we omit many fields like icmp-type from examples to avoid clutter.

Even though our technique can be applied to chains with default ACCEPT policy, all examples and discussions assume that the chains have default REJECT policy. Note that the chain name is shown above the rules in the tabular format to make the tables more understandable. In practice, we generate different rule sets for different built-in chains.

Priority Elimination Phase

In this phase we take the rule set with priorities and generate *flattened* rule set. Flattened rule set contains rules which have no priority relation so they can be arbitrarily reordered and merged to generate a compact policy. The idea behind flattening of rules is simple. Consider a rule set RS with rules R_i such that priority of R_i is higher than the priority of R_j iff $i < j$. The semantics of such a *prioritized* rule set are that a packet is matched by a rule R_j iff it satisfies the matching criteria of R_j and doesn't satisfy the matching criteria of any of the rules R_i such that $i < j$. In other words, a packet can match a rule only if it is not matched by any higher priority rule. For example, R_3 can match a packet only if R_1 and R_2 do not match it. Thus a prioritized rule set can be converted to a flattened rule set by replacing R_j by $\bigwedge_{i=1}^{j-1} \neg R_i \wedge R_j$, $2 \leq j \leq n$ where n is the total number of rules in the set. In our example, R_1 will not be modified while R_2 will be replaced by $\neg R_1 \wedge R_2$ and R_3 by $\neg R_1 \wedge \neg R_2 \wedge R_3$.

The problem with the naive way of generating flattened rules is that it can lead to exponential number of rules. In [11], we developed a way of creating a directed acyclic graph (DAG) called *packet classification automaton* that avoids this exponential blowup. We use the packet classification automaton to generate

flattened rule set. Here we describe the characteristics of the automaton without going into the details of the construction algorithm; which can be found in [11].

- each node (except the final nodes) is annotated with a packet header field. This denotes that the node performs a test on that field.
- the leaf nodes correspond to the action to be performed when a packet is matched by a rule. For example, for *iptables* the leaf nodes correspond to the higher level actions allow and deny. We map targets like ACCEPT, QUEUE, and LOG to allow and REJECT and DROP to deny. A path from the root to a leaf represents the tests to be performed on a packet to match it against the rule set and the action to be taken on the packet.
- at each node the outgoing edges are labeled with the different values specified for the packet header field specified on the node.
- at each node there is an additional outgoing edge called as “else” edge. This edge is taken when a packet has a field value different from any of the values listed in the other outgoing edges from that node.
- nodes at the same height in different subgraphs can have tests for different fields.

This automaton has the following interesting properties:

- **Property 1** *Packet classification automaton is equivalent to the prioritized rule set, i.e., any packet that is allowed/denied by the rule set has a path from root to allow/deny node in the automaton and vice versa.*
- **Property 2** *If the input rule set is comprehensive, i.e., for every packet there is a rule in the rule set that matches it, then the automaton has a path from root to a leaf for every packet. Moreover, the path from root to leaf is unique.*

We generate flattened rule set by considering all paths from root to the leaves in the graph. Each path corresponds to a rule in the flattened rule set. Consider a *iptables* script with the three rules for FORWARD chain shown in Listing 1.

```
1. IPTABLES -A FORWARD -s 192.168.1.0/24 -d 120.240.18.1 --dport 25 -j ACCEPT
2. IPTABLES -A FORWARD -s 120.240.18.1 -d 120.240.20.0/24 --dport 25 -j ACCEPT
3. IPTABLES -A FORWARD -j REJECT
```

Figure 3: *iptables* rule set 1.

#	shost	sport	dhost	dport	target
FORWARD (Default: Reject)					
1	192.168.1.0/24	*	120.240.18.1	25	ACCEPT
2	120.240.18.1	*	120.240.20.0/24	25	ACCEPT

Table 1: *iptables* rules in Figure 3 represented in tabular format.

```
-d 192.168.1.1 -s 192.168.1.3 --dport 22 -j ACCEPT
-d 192.168.1.2 -s 192.168.1.4 --dport 22 -j ACCEPT
-j REJECT
```

Listing 1: Sample *iptables* rule set as input.

Figure 6 shows the packet classification automaton for this sample rule set. Here each node is labeled with the packet field being tested at that node.

Pruning

Figure 4 shows the packet classification automaton for a firewall rule set with 65 rules for a small network within our department. The bottom row has two leaf nodes corresponding to the actions: allow and deny. We can see that even for small sized rule set, there are a large number of paths in the graph.

We prune this graph to reduce the number of paths that we have to consider. The following are the steps that we perform for pruning this graph.

1. We remove deny node and all incoming edges to it. If this results in an intermediate node becoming leaf node, then we remove that node and its incoming edges. We recursively do this for all *ancestors* of deny except the root node. Figure 7 shows the graph after deny node has been removed from the graph in Figure 6. Now the graph contains only paths from root to allow.

generate from this graph is no longer *comprehensive*. But this problem can be easily solved by having a default reject policy for the flattened rules, i.e., packets that are not matched by any rule in the flattened rule set are discarded.

2. We do a bottom-up traversal of the graph and merge equivalent states. We consider two states r and s as equivalent if they have transitions to the same state t_i on label l_i for all outgoing edges. For the graph in Figure 7, both the dport nodes have an edge to allow for value 22. These nodes are merged to get a graph as shown in Figure 8.
3. The previous two steps create new opportunities for reducing the number of paths through the graph. We can now merge multiple edges which connect the same nodes. In the case where the edge merging involves else edge, the merged edge is labeled with else. A special case of this is when all outgoing edges from a node can be merged with else edge. In this case the node with the merged outgoing edges is removed as the merged edge indicates that this

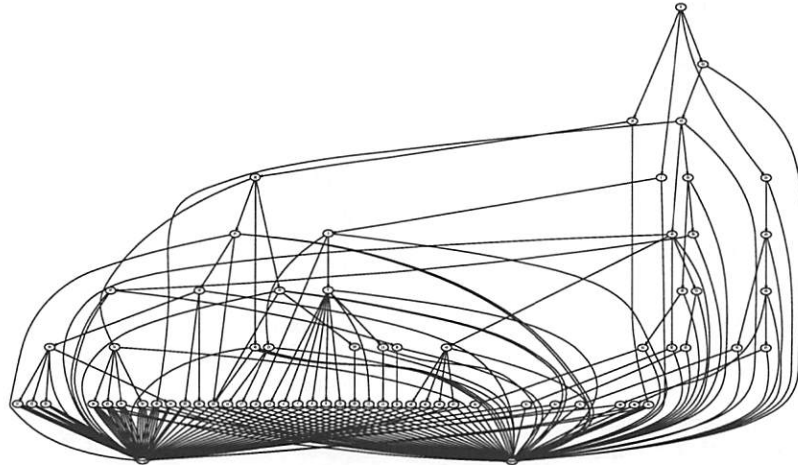


Figure 4: Initial graph for network in the department.

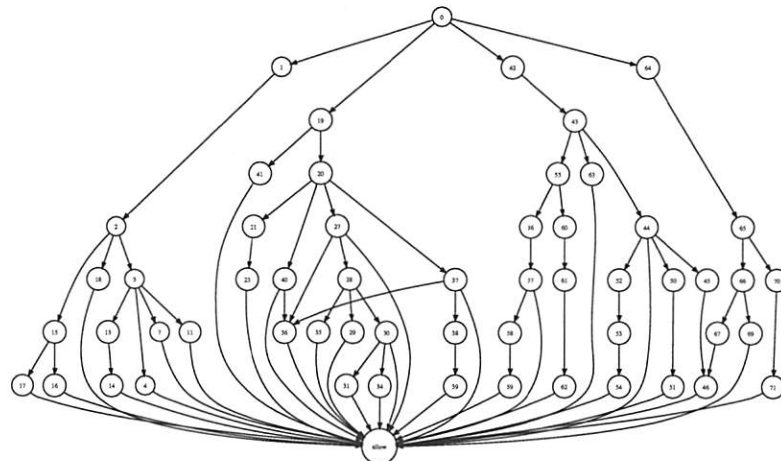


Figure 5: Pruned graph for graph in Figure 4.

transition is taken irrespective of the value of the field in the removed node. This edge merging is also done in a bottom up fashion.

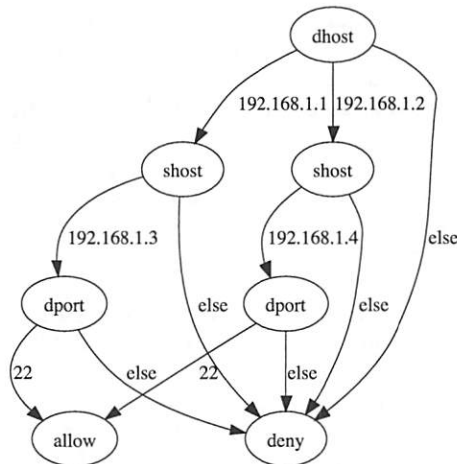


Figure 6: Unpruned graph for sample rules.

Figure 5 shows the pruned graph corresponding to the graph in Figure 4. We can read off all the paths from the root to accept to get flattened rule set.

Policy Inference Phase

As the main goal of our research was to come up with a compact representation of the rules, we asked ourselves the following questions:

- how can we compare two representations of the same rule set?
- how can we reduce the complexity of the flattened rules?

In this section we present our answers to these questions.

Complexity

We call a particular representation of rules as *policy*. Intuitively, a policy that requires a bigger description is more complex. For example, consider an organization which has 192.168.1.0/24 as the internal network. It has rules that allow auth packets to all hosts in the network and ssh packets only to the hosts 192.168.1.5, 192.168.1.6, and 192.168.1.7. These rules can be represented as *Policy 1* as shown in Listing 2. A more compact way of representing these rules (*Policy 2*) is shown in Listing 3.

We capture this notion of how complicated a policy is by the following definition:

```
Accept packets
a. TO [192.168.1.0 - 4], [192.168.1.8 - 255] FOR auth
b. TO [192.168.1.5 - 7] FOR auth, ssh
```

Listing 2: Policy 1.

```
Accept packets
a. TO 192.168.1.0/24 FOR auth
b. TO [192.168.1.5 - 7] FOR ssh
```

Listing 3: Policy 2 – More compact version of policy in Listing 2.

Definition 3: Complexity of a policy

- **Data item** is any value that is used in a rule.
- **Complexity of a rule** is the number of data items present in the rule.
- **Complexity of a policy** is the sum of the complexity of all rules in the policy.

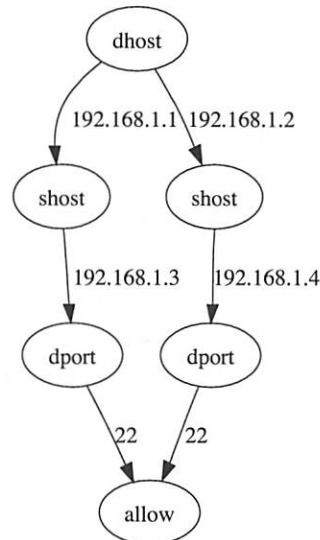


Figure 7: Graph with deny node removed from the graph in Figure 6.

Data items refer to the different values of packet fields present in the policy. For example, in the policy given above, the data items are 192.168.1.0/24, [192.168.1.5 - 7], auth, and ssh. For the earlier policy the data items are [192.168.1.0 - 4], [192.168.1.8 - 255], [192.168.1.5 - 7], auth, and ssh. The complexity of rules in *Policy 1* is 3 for rule *a* and 3 for rule *b*. We note that ranges such as [192.168.1.5 - 7] are treated as a single data item and hence contribute 1 to the complexity of a rule. Even though our examples contain * for certain fields, they are there just to increase readability. Our final policy (as shown in Figure 2) does not contain any “*”. Therefore, “*” doesn’t contribute anything to the complexity of a rule. Similarly, in *Policy 2* the complexity of rules *a* and *b* are 2 each. The complexity of *Policy 2* is 4. This is less than that of *Policy 1* which is 6. Hence we can say that *Policy 2* is a more compact representation than *Policy 1*.

Problem Statement

We describe the problem statement in this section. At the end of *priority elimination* phase we have

a large number of rules. We want to merge the rules in such a way that the complexity of the generated rule set is minimum. We can merge rules which have the same values for all but one field by performing a *union* operation over the field which has differing values in the rules. The main issue with merging rules is that different subsets of rules can be merged on different fields. So we need to select the subsets in such a way that merging them leads to minimum complexity of the entire rule set. This problem can be illustrated

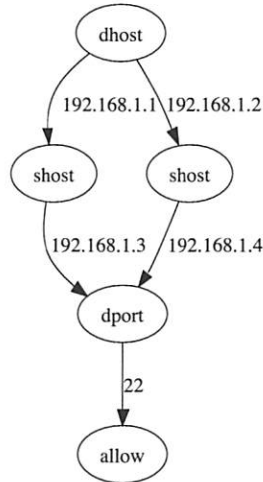


Figure 8: dport node merged from graph in Figure 7.

by considering the rule set shown in Table 2. We can merge rules *b* and *c* on *shost* to get a new rule $\{bc\}$. Note that we label a merged rule by concatenating the labels of the original rules. We enclose the new label in *braces* to indicate how the rules merged.⁴ We can then merge $\{bc\}$ with *a* to get rule $\{a\{bc\}\}$. This gives the following policy (*Policy 3*) which has a complexity of 12; see Listing 4.

⁴The notation $\{bc\}$ is different from $\{b, c\}$ which means a set containing the rules *b* and *c*

```
Accept packets
{a{bc}}. 192.168.1.1, 192.168.1.10 TO 192.168.2.1 FOR http, smtp (5)
d.      192.168.1.10 TO 192.168.2.2 FOR smtp (3)
e.      192.168.1.10 TO 192.168.2.1, 192.168.2.2 FOR ssh (4)
```

Listing 4: Policy 3.

#	shost	dhost	dport	target
a	192.168.1.1, 192.168.1.10	192.168.2.1	80	ACCEPT
b	192.168.1.1	192.168.2.1	25	ACCEPT
c	192.168.1.10	192.168.2.1	25	ACCEPT
d	192.168.1.10	192.168.2.2	25	ACCEPT
e	192.168.1.10	192.168.2.1, 192.168.2.2	22	ACCEPT

Table 2: Sample flattened rule set.

```
Accept packets
{a{bc}}. 192.168.1.1, 192.168.1.10 TO 192.168.2.1 FOR http, smtp (5)
{{cd}e}. 192.168.1.10 TO 192.168.2.1, 192.168.2.2 FOR smtp, ssh (5)
```

Listing 5: Policy 4.

The complexity of each rule is written in parenthesis besides the rules. Since all the rules in the flattened rule set have the same action, we can even generate overlapping rules to get a more compact policy. We can merge rules *a*, *b*, and *c* as before to get rule $\{a\{bc\}\}$, and also merge *c* and *d* on *dhost* followed by $\{cd\}$ with *e* on *dport*. Listing 5 shows policy (*Policy 4*), with complexity 10, that is obtained by merging the rules in this way.

The Policy Complexity Definition is tied to the way the rules are represented. Thus, the interesting question is *given a rule set, can we find an equivalent representation which enforces the same security policy but has lower complexity?* Our goal is to find a *representation of the rule set that implements the same policy and has minimum complexity*. A natural way to select the policy with minimum complexity is to assign weights based on the complexity to all subsets of the flattened rule set and select a minimum weight set cover. The problem of determining the minimum weight set cover is NP-complete. However, we have found that fairly simple methods, based on exploiting the structure of the flattened rules, yield good results for finding a policy with low complexity.

Computing Weight of Subsets of Rules

Computing the complexity of all subsets of a rule set is also very hard. To see this, consider that we have a set of *n* rules. We want to find a policy with minimum complexity for this rule set. To find the complexity of a set containing *n* - 1 of these rules is similar to the original problem.

In practice, we can avoid this problem as we do not need to generate all subsets of rules in the original rule set. To understand this consider original rule set $\{a, b, c\}$ such that rules *a* and *b* can be merged on certain field. Now the *weight* of $\{a, b\}$ is the complexity of the merged rule $\{ab\}$. But the *weight* of $\{b, c\}$ is the sum of the complexities of *b* and *c*. So we do not need

to consider the *weight* of $\{b, c\}$ if we have the *weights* for $\{b\}$ and $\{c\}$. This leads us to conclude that we need to consider only the subsets of the original set in which each rule merges with some other rule or with a new merged rule. We maintain the subsets of rules that we need to consider in a *working set*.

Here we describe the algorithm shown in Figure 9 to generate the working set. Initially we have a set \mathcal{R} of flattened rules $\{r_1, r_2, \dots, r_n\}$. We want to generate a set \mathcal{W} that contains the subsets of \mathcal{R} that we need to consider for the minimum weight set cover problem. We start by putting all singleton subsets of \mathcal{R} in \mathcal{W} . For example in Figure 2,

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4\}.$$

Note that the value after “:” is the *weight* of the corresponding set. For singleton sets, the *weight* is the same as the complexity of the rule. Now we compare each element in \mathcal{W} with the other elements in \mathcal{W} . If the two elements under consideration can be merged, then we merge them and add the new merged rule to the working set. Now,

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4, \{bc\}:4, \{cd\}:4\}.$$

We continue this process of merging and adding new rules till no more rules can be added to \mathcal{W} . In our example, $\{bc\}$ can be merged with $\{a\}$ and $\{cd\}$ with $\{e\}$. After this no more merging is possible. So the final set is

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4, \{bc\}:4, \{cd\}:4, \{a\{bc\}\}:5, \{\{cd\}e\}:5\}.$$

Merge Graphs

Comparing each element in \mathcal{W} with all the other elements is computationally expensive. We overcome this problem by generating a graph, which we call as *merge graph*, that allows us to avoid many comparisons. *Merge graph* is an acyclic graph which initially contains nodes corresponding to the flattened rules and no edges. As we merge rules, we add nodes corresponding to the merged rules and edges from the new node to the constituent rule nodes. For example, after we add $\{bc\}$ to \mathcal{W} , we can represent \mathcal{W} as a *merge*

graph as shown in Figure 10. The *merge graph* may contain disconnected subgraphs. For each node in the

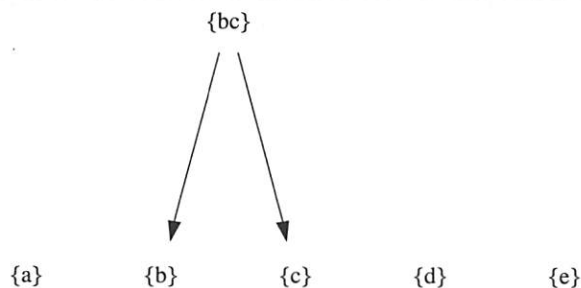


Figure 10: Merge graph after adding $\{bc\}$.

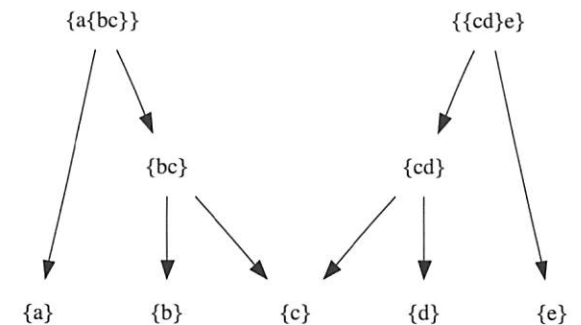


Figure 11: Final merge graph.

graph, we compare it with only the nodes in other disconnected components. This way we avoid redundant comparisons. In Figure 10, we compare the rule $\{bc\}$ with $\{a\}$, $\{d\}$, and $\{e\}$. This avoids redundant comparisons with $\{b\}$ and $\{c\}$. For large rule sets with multiple rules that merge, this optimization proves very useful. Figure 11 shows the final *merge graph* for our example.

Solving Minimum Weight Set Cover

Now that we have the working set we can find the minimum weight set cover to get a policy with low complexity. Conceptually we can think of each element of \mathcal{W} as a set containing the rules that have been merged to form that element. For the example in the previous section,

```

1. procedure GenerateWorkingSet( $\mathcal{R}$ ) {
2.    $\mathcal{W} = \emptyset$ 
3.   for  $i = 1$  to  $|\mathcal{R}|$  do           /*  $\mathcal{R} = \{r_1, r_2, \dots, r_n\}$  */
4.      $\mathcal{W} = \mathcal{W} \cup \{r_i\}$          /* add all flattened rules to the working set */
5.   end
6.   for  $i = 1$  to  $|\mathcal{W}|$  do
7.     for  $j = 1$  to  $|\mathcal{W}|$  do
8.        $S = \text{MergeRules}(w_i, w_j)$  /*  $w_i, w_j \in \mathcal{W}$  */
9.       if  $S \neq \emptyset$ 
10.         $\mathcal{W} = \mathcal{W} \cup S$           /* add the new merged rule to the working set */
11.      endif
12.    end
13.  end
14.  return  $\mathcal{W}$ 
15. }
```

Figure 9: Algorithm for Constructing Working Set.

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4, \{b,c\}:4, \{c,d\}:4, \{a,b,c\}:5, \{c,d,e\}:5\}$$

Our target is to find a set cover $C = \{c_1, c_2, \dots, c_k\}$, i.e., $C \subseteq \mathcal{W} \wedge \bigcup_{i=1}^k c_i = \mathcal{R}$ such that $\sum_{i=1}^k \text{weight}(c_i)$ is minimum.

Figure 12 shows our algorithm based on greedy heuristics to solve the problem. We use a set, \mathcal{A} , to keep track of the rules that are covered as the set cover C is built. For each set in \mathcal{W} , we define,

$$\text{cost}(w_i) = \frac{\text{weight}(w_i)}{|w_i/\mathcal{A}|}$$

where $\text{cost}(w_i)$ represents the cost incurred (in terms of *weight*) per new rule that will be covered by including a set w_i in the set cover. In each iteration, we pick a set w_i that has the lowest cost (step 6 in our algorithm). In our example, initially the *cost* is $4/1 = 4$ for $\{a\}$, $4/2 = 2$ for $\{b, c\}$, $5/3 = 1.67$ for $\{a, b, c\}$ and so on. In case of a tie, we pick the set with higher cardinality. This algorithm returns $C = \{\{a, b, c\}, \{c, d, e\}\}$ as the minimum weight set cover. We know that these sets correspond to merged rules $\{a\{bc\}\}$ and $\{cd\}e\}$. So we can represent the rules in the example as *Policy 4* as shown in the previous section.

Related Work

There are many tools (e.g., Firmato [2], Shorewall [4], Firestarter [3]) that are available for generating low-level firewall rules from high-level policy. Our technique can be used in conjunction with these tools to help refactoring. These tools can be used to generate firewall rules from scratch. If our technique is combined with these tools then we can use these tools to make changes to existing low level rules.

Fang [7] and ITVal [9] are tools that provide querying facility. But it puts the onus on the system administrator to figure out what queries to perform. Lumeta Firewall Analyzer [12] solves this problem by querying the system for all packets that can be accepted. The problem with this approach is that this results in a large amount of data being presented to the user. In contrast, we try to present the result of our

analysis in a compact fashion to the administrator. Moreover, it is easy to provide querying capability using our technique.

Yuan, et al. [13], Gouda, et al. [6], Al-Shaer, et al. [1] have looked at the problem of identifying configuration errors in the firewall rules. The problem with these approaches is that the administrator has to decide whether the alert generated by these tools are due to rules that are put in intentionally or unintentionally. Our technique can help in solving this problem by providing a high level view of the security policy.

Marmorstein, et al. [8] generate policy by grouping similar hosts. Our work is more general in the sense that we can group together arbitrary things to generate more compact representation. Golnabi et al. [5] have looked at the problem of generating high level policy. But their approach is based on data mining of firewall logs while we try to extract the policy from the rules itself.

Conclusions

In this paper, we presented a new technique for extracting high-level security policy from low-level rules. Unlike previous techniques, our technique generates policy which is compact. This will help system administrators to understand the existing low level rule sets and encourage them to refactor the low-level rules instead of making small changes to the rules set when requirements change. This will make the rule sets more manageable and will likely result in reducing the errors in configuration of firewalls. We also presented a way for comparing whether one policy representation is better than another. In our preliminary experiments, we obtained 50 flattened rules with 197 data items after the *priority elimination* phase from a 65 rule firewall. The final inferred policy on the other hand had just 21 rules with 129 data items. These results indicate that we can generate high level policy which is easier to understand and manage.

Acknowledgments

We would like to thank Mayur Mahajan for his help in developing the graph library used for generating

```

1. procedure MinimumWeightSetCover( $\mathcal{R}, \mathcal{W}$ ) {
2.    $\mathcal{A} = \emptyset$                                      /*  $\mathcal{A} \subseteq \mathcal{R}$  is the set of covered elements */
3.    $C = \emptyset$                                    /*  $C \subseteq \mathcal{W}$  is the solution set */
4.   while  $\mathcal{A} \neq \mathcal{R}$  do                           /*  $\mathcal{A}$  is not a set cover */
5.     for  $i = 1$  to  $|\mathcal{W}|$ 
6.       compute  $\text{cost}(w_i)$ 
7.     end
8.     choose  $w_i$  with minimal  $\text{cost}(w_i)$ 
9.      $\mathcal{A} = \mathcal{A} \cup w_i$                              /* rules in  $w_i$  are covered */
10.     $C = C \cup \{w_i\}$                              /* add  $w_i$  to solution set */
11.     $\mathcal{W} = \mathcal{W} / \{w_i\}$                            /* remove  $w_i$  from working set */
12.  end
13.  return  $C$ 
14. }
```

Figure 12: Algorithm for Minimum Weight Set Cover.

the flattened rules and for his work on pruning the graphs. We would like to thank Weiqing Sun, Lorenzo Cavallaro, and the anonymous reviewers for their insightful comments.

This material is based upon work supported by the NSF grants 0627687 and 0208877. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

Author Biographies

All The authors of this paper are members of the Secure Systems Laboratory of Stony Brook and their homepages are accessible on the web from the laboratory page at <http://www.seclab.cs.sunysb.edu>.

R. Sekar is currently Professor of Computer Science and heads the Secure Systems Laboratory at Stony Brook University. Prof. Sekar's research interests include computer system and network security, software and distributed systems, programming languages and software engineering. He can be reached by email at sekar@cs.sunysb.edu.

Alok Tongaonkar is a Ph.D. student in the CS department at Stony Brook. His main research area is computer security and is currently working on analyzing and optimizing firewalls and intrusion detection systems. He is available at alok@cs.sunysb.edu.

Niranjan Inamdar is a M.S. student in the CS department at Stony Brook. Niranjan does research in the area of computer security. He can be reached via email at niranjan@cs.sunysb.edu.

Bibliography

- [1] Al-Shaer, Ehab and Hazem Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *IEEE INFOCOM*, 2004.
- [2] Bartal, Yair, et al., "Firmato: A Novel Firewall Management Toolkit," *IEEE Security and Policy*, 1999.
- [3] *Firestarter*, <http://www.fs-security.com>.
- [4] *Shorewall Firewall*, <http://www.shorewall.net>.
- [5] Golnabi, Korosh, et al., "Analysis of Firewall Policy Rule using Data Mining Techniques," *IEEE/IFIP Network Operations and Management Symposium*, 2006.
- [6] Gouda, Mohamed G. and Alex X. Liu, "Firewall Design: Consistency, Completeness, and Compactness," *International Conference on Distributed Computing Systems*, 2004.
- [7] Mayer, Alain, et al., "Fang: A Firewall Analysis Engine," *IEEE Symposium on Security and Privacy*, 2000.
- [8] Marmorstein, Robert and Phil Kearns, "Firewall Analysis with Policy-Based Host Classification", *20th Large Installation Systems Administration Conference*, 2006.

- [9] Marmorstein, Robert and Phil Kearns, "A Tool for Automated Iptables Firewall Analysis," *Freenix Track, USENIX Annual Technical Conference*, 2005.
- [10] Rubin, Aviel, Dan Geer, and Marcus Ranum, *Web Security Sourcebook*, Wiley Computer Publishing, 1997.
- [11] Tongaonkar, Alok, "Fast Pattern-Matching Techniques for Packet Filtering," *Master's Thesis Report, Stony Brook University*, 2004, <http://www.seclab.cs.sunysb.edu/seclab/pubs/theses/alok.pdf>.
- [12] Wool, Avishai, "Architecting the Lumeta Firewall Analyzer," *10th USENIX Security Symposium*, 2001.
- [13] Yuan, Lihua, et al., "Fireman: A Toolkit for Firewall Modeling and Analysis," *IEEE Symposium on Security and Privacy*, 2006.
- [14] Eychenne, Herve, *iptables Man Page*, March, 2002.

Assisted Firewall Policy Repair Using Examples and History

Robert Marmorstein and Phil Kearns – The College of William & Mary

ABSTRACT

Firewall policies can be extremely complex and difficult to maintain, especially on networks with more than a few hundred machines. The difficulty of configuring a firewall properly often leads to serious errors in the firewall configuration or discourage system administrators from implementing restrictive policies.

In previous research, we developed a technique for modeling firewall policies using Multiway Decision Diagrams and performing logical queries against a decision diagram model. Using the query logic, the system administrator can detect errors in the policy and gain a deeper understanding of the behavior of the firewall. The technique is extremely efficient and can process policies with thousands of rules in just a few seconds. While queries are a significant improvement over manual inspection of the policy for detecting that errors exist, they provide only limited assistance in repairing a broken policy. In this paper we present two extensions to our work, examples and history, which enable the administrator to more easily repair a policy which contains errors.

An example is a representative packet which illustrates that the firewall complies with or (more importantly) deviates from its expected behavior. History records the specific rules involved in the deviation. Examples and history provide guidance in finding and fixing faults in a firewall rule set. These contributions can be also be used with the equivalence class analysis to reduce the burden of designing a complicated set of assertions.

Introduction

The administrator who maintains a restrictive firewall policy on a large network must spend a considerable amount of time and effort updating and testing the filtering rules. Requests for new services, changes in the physical topology of the network, and the emergence of new security threats require continual modification of the policy. As the policy changes and grows, it can be difficult for the administrator to avoid introducing errors into the rule set. Repairing these errors is often very challenging. Firewall policy errors are subtle and difficult to detect. Even when the existence of an error is obvious, discovering the source of the problem and correcting it can be tedious and expensive.

In previous work, we introduced techniques for quickly and easily validating a firewall policy using logical queries against a Multiway-Decision Diagram model of the firewall policy. The MDD approach is very efficient (complex queries involving rule sets with hundreds of rules usually take only a few seconds) and allows for very flexible identification of errors. Like most existing approaches, the MDD query technique addresses the issue of testing the firewall for errors, but leaves the problem of repairing the policy entirely up to the administrator. Because tracing through dozens or perhaps hundreds of correct rules to find the two or three critical inconsistencies can take hours or even days, this is a significant burden for administrators of a large network.

Chain Forward (Default Drop)					
#	Target	Source	Destination	Interface	Flags
1	DROP	192.168.1.0/24	anywhere	!eth2	
2	DROP	192.168.3.0/22	192.168.2.0/24	any	
3	ACCEPT	anywhere	192.168.2.4	any	dpt:tcp 80
4	DROP	anywhere	192.168.2.0/24	any	
5	ACCEPT	192.168.1.0/24	anywhere	any	

Figure 1: A rule set which incorrectly blocks access from the 192.168.1.0/24 subnet. Rule 1 of the policy ensures that traffic from the 192.168.1.0/24 subnet arrives on the correct interface. Rule 2 blocks traffic from the insecure wireless network to the server subnet. Rule 3 grants HTTP access to the web server to appropriate hosts. Rule 4 prevents external access to other servers. Rule 5 allows hosts on the trusted subnet to transmit packets that have not been blocked by some previous rule.

In this work, we present two novel techniques that enable “directed repair” of the firewall policy. Using these techniques, the system administrator not only can identify the existence of an error in the policy, but can trace it back to its root causes without an expensive manual inspection of the rule set. The first major contribution is a technique for providing example packets that illustrate that the firewall violates a set of security requirements. The second contribution creates a history map which identifies the particular firewall rules which cause the firewall to deviate from its desired behavior.

While these techniques do not fully automate the process of repairing the firewall, they do provide the system administrator with information that makes repair much easier than a simple verification of the policy. We have implemented both techniques in ITVal [6], a firewall analysis and repair tool for iptables firewalls. Although we use the Linux iptables firewall for the examples in this paper, it is possible to adapt these techniques to work with other platforms such as PIX and Checkpoint firewalls. There are also several fairly effective scripts for converting ipfw and ipchains firewall to iptables syntax [11] which can be used to adapt such policies to a format compatible with ITVal.

The remainder of this paper is structured as follows. The next section describes the difficulties which a system administrator encounters in repairing a firewall. Then we discuss partially automated repair of the policy. The next two sections detail our techniques for generating examples and history, respectively. Sections on implementation using MDDs and a description how this work can be combined with our previous work on equivalence class analysis of a firewall policy follow. Finally, we discuss related work and make a few concluding remarks.

Firewall Policy Errors

The techniques discussed in our previous work allow a system administrator to perform basic logical queries against a firewall policy using a simple specification language. For instance, to ask which hosts can access a web server, host 192.168.2.4, the administrator can use the query `QUERY SADDY TO 192.168.2.4 AND FORWARD ACCEPTED`; which will list the source addresses of any host that can access the web server without being blocked by the firewall. Inspecting this list of addresses may allow the user to detect an error in the configuration of the firewall. If the address of a malicious host appears in the list, for instance, it is clear that there is a problem with the firewall policy.

Queries allow the system administrator to identify many serious errors in the configuration of a firewall, but provide only a limited amount of information about each error. For instance, if the system administrator uses the query “Which hosts can connect to the mail server?” to determine whether the firewall blocks

external hosts, the analysis engine will list those hosts that have unwanted access to the server, but will not provide any additional information that can be used to understand why the firewall failed to prevent access. It may be that the error only occurs when connections are made on a particular network interface or by a particular network protocol. Access to this information could greatly assist the system administrator in repairing the policy, but traditional testing tools do not provide these helpful clues. Another way to think about this is to say that a query engine discovers an error (the ultimate consequence of a problem in the policy), but not the fault (the mistake in the rules that causes the error).

Sometimes, helpful information can be obtained using additional queries or by refining the query to provide more information. Often, however, the process of developing a sufficiently detailed set of queries requires almost as much effort as manual repair of the policy.

This means that query tools are usually limited to detecting whether an error exists and have only limited utility in guiding repair of the policy. To repair the policy by hand, the system administrator must carefully consider each filtering rule to determine whether it is relevant to the error and, if so, whether it is correct. Since most of the rules will usually be either irrelevant or valid, manual repair is a very inefficient and time consuming process, especially when an error has many potential causes.

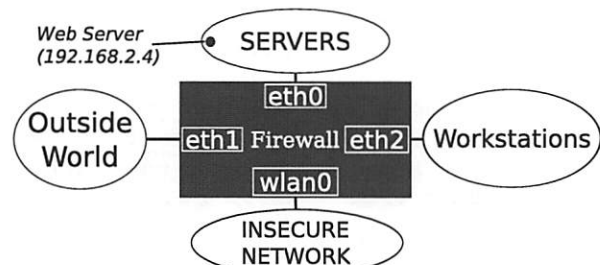


Figure 2: A typical firewall, which protects hosts on two subnets against intrusions from a third, untrusted network and the outside world. One of the protected subnets contains a web server, host 192.168.2.4, to which remote connections are allowed.

Figure 1 shows how difficult it can be to trace a firewall error to its source. This rule set protects workstations on the subnet 192.168.1.0/24 and servers on the 192.168.2.0/24 subnet against attacks from the outside world and an insecure wireless network on the 192.168.3.0/24 subnet. An illustration of the network is given in Figure 2.

The system administrator wants to allow access to the web server, host 192.168.2.4, from any system in the outside world except those on the unsecured wireless network. All other external traffic to the web server should be blocked.

It is fairly easy to determine that the rule set fails to enforce these requirements. If the system administrator opens up a web browser and tries to connect to the web server from a host on the trusted network, the firewall will refuse to allow the connection. Discovering the cause of this error is more challenging, since nearly every rule of the policy plays some role in the filtering decision. An error in rule 4, which drops traffic to the protected subnet, could be the source of the error. A typo in rule 3, which overrides rule 4 to allow web traffic to enter the network might be another the cause. Rule 1, an anti-spoofing rule which blocks traffic from the “wrong” interface, might also be to blame.

As it turns out, the fault that produces the observed error is in rule 2. An incorrect subnet mask in rule 2 causes the firewall to block traffic from the protected network as well as the untrusted net. Manual analysis of the policy requires a careful and tedious inspection of every rule in the policy to identify this fault. For the five rule policy shown here, this inspection might not take too long. However, a policy with more than a few dozen rules would be much more difficult to analyze. Partially automating the repair process in a way that narrows down the potential sources of the error to just one or two rules could save the administrator a significant amount of effort.

Partially Automated Firewall Repair

Unfortunately, it is impossible to fully automate repair of a generic firewall policy because incorrect behavior on one network may be expected behavior on another. For instance, on one network it may be desirable to allow SMTP traffic to reach certain hosts, such as the mail servers. On another network, however, a policy that permits SMTP traffic may spam-bots to compromise important systems. Without input from the user, a repair algorithm cannot distinguish between these two cases.

While a fully automatic strategy for firewall repair is impossible, partial automation is possible. Gouda, Liu, et al. have done significant work on repair of structural errors in the firewall policy [4]. Their technique uses transformation of decision diagrams to produce an improved rule set in which problems such as shadowed or duplicate rules have been eliminated. This strategy does not require any assistance from the user. Unfortunately, these techniques do not address repair of logical errors such as typos or out-of-order rules.

Another approach is to allow the user to make the final decision about how to repair the policy, but automate the process of finding the faults responsible for the error. By providing the system administrator with sufficient information about the possible causes of the error, we can guide her toward a few possible solutions, from which she can choose the one best suited to her network. This “directed repair” of the policy alleviates much of the tedious work required to find faults and fix the policy.

Directed Repair

In previous work [7, 8, 9], we explored ways to detect errors in a firewall configuration using logical queries and an equivalence class decomposition of the network. In this paper, we describe two extensions of this work that enable directed repair of the firewall policy. One technique generates relevant counterexamples from which the system administrator can obtain detailed information about security failures in the policy. The second technique provides an extensive “history analysis” that identifies potential sources of the error and lists rules which should be considered for modification. The history analysis can also be used with the equivalence technique described in [9], which addresses the need for extensive preparation of logical queries. We implement both of these techniques as extensions to ITVal [6], an open source firewall testing tool developed as part of our previous work.

```

FROM <address_range>
    matches all packets with source address in address_
    range.
TO <address_range>
    matches all packets with destination address in
    address_range .
ON <port_range>
    matches all packets with source port in port_range.
FOR <port_range>
    matches all packets with destination port in port_
    range
IN <s>
    matches all packets associated with connections in
    state s
WITH <flag_set>
    matches all packets with the TCP flags in flag_set
    enabled
ACCEPTED <chain>
    matches all packets accepted by built-in chain chain
DROPPED <chain>
    matches all packets rejected by built-in chain chain
INFACE <iface>
    matches all packets received by network interface
    iface
OUTFACE <iface>
    matches all packets transmitted on network inter-
    face iface

```

Figure 3: ITVal primitives.

To use these techniques, the user specifies the desired behavior of the firewall using logical assertions. The syntax for assertions is derived from the query language explained in [8]. The right and left conditions of the assertion are built from a set of simple primitives such as those in Figure 3, which can be combined using the logical operators AND, OR, and NOT to create complex conditions describing sets of packets whose treatment by the firewall requires analysis. For example, we can describe all accepted SSH

packets from subnet 192.168.1.0/24 on interface eth0 using the condition

```
FOR TCP 22 AND
FROM 192.168.1.* AND
INFACE eth0 AND
  (ACCEPTED FORWARD OR
   ACCEPTED INPUT);
```

The user can construct two types of assertions from these conditions. Equality assertions have the form:

```
ASSERT <A> IS <B>
```

where A and B are conditions. Containment assertions have the form

```
ASSERT <A> SUBSET OF <B>
```

Equality assertions specify that those packets which match condition A are exactly those that match condition B. Containment assertions specify that the set of packets that satisfy condition A is (non-strictly) contained in the set of packets that satisfy condition B. Using these assertions, the user can describe important high-level security invariants which the policy should always satisfy.

For instance, the containment assertion

```
ASSERT FROM 192.168.3.*
SUBSET OF DROPPED FORWARD;
```

specifies that any packet from subnet 192.168.3.0/24 is dropped. The equality assertion

```
ASSERT FROM 192.168.2.*
IS (FOR TCP 80
AND ACCEPTED FORWARD);
```

can be used to check that only HTTP packets are allowed to enter the network from the 192.168.2.0/24 subnet and that no other web connections are allowed by the firewall. We call the set of packets that match a condition its match set and the set of packets that cause an assertion to fail the assertion's fail set. Assertions provide many advantages over simple queries. While queries allow the user to obtain a significant amount of information about the policy, a query does not provide the analysis engine with any description of the expected behavior of the firewall. Therefore, using assertions enables the engine to provide more useful and relevant output.

Counterexamples and Witnesses

One useful advantage of assertion analysis is that it allows generation of relevant counterexamples. These

counterexamples provide a context for the error which can often help the administrator discover why a failure has occurred.

The example policy in Figure 4 isolates an untrusted research network 192.168.2.0/24 from the outside world. SSH traffic from the untrusted network to hosts on subnet 192.168.1.0/24 is permitted, but all other traffic from the network is denied. The 192.168.1.0/24 subnet contains several world-accessible web servers to which the policy grants access. However, the rule set blocks connections from 63.118.7.16, a malicious host. Trusted hosts are allowed to make connections to the web servers and an external server, host 131.106.3.253, but cannot make any other connections.

To test whether the untrusted hosts are sufficiently restricted by the firewall, the administrator uses the assertion

```
ASSERT (FROM 192.168.2.*
AND NOT FOR TCP 22)
SUBSET OF DROPPED FORWARD;
```

which specifies that only SSH traffic is accepted from hosts on the untrusted network. Due to an error in the ordering of rules 2 and 4, the assertion will fail. This subtle error could be very difficult to detect in a lengthier policy in which the rules were much further apart. Using ITVal, the administrator can easily discover that the assertion fails. Knowing that the assertion does not hold is an important first step, but does not give much information about the cause of the error. To give the user more information about the source of the error, we generate a counterexample – a packet that demonstrates the falsity of the assertion. Figure 5 shows the generation of one possible counterexample. The user specifies that an example should be generated by inserting the keyword EXAMPLE at the beginning of the assertion.

Examination of the counterexample gives the system administrator important information about the assertion failure. One significant clue is that the example packet arrived on interface eth1. Since only rule 2 mentions eth1, this fact draws the administrator's immediate attention to the rule ordering error, which can now be corrected by moving rule 2 to the correct location in the policy.

Sometimes it is desirable to obtain an example even when an assertion succeeds. We call such an

Chain FORWARD (Default DROP)					
#	Target	Source	Destination	Interface	Flags
1	ACCEPT	anywhere	192.168.1.0/24	eth0	dpt:tcp 22
2	ACCEPT	anywhere	131.106.3.253	eth1	
3	DROP	63.118.7.16	anywhere	eth0	
4	DROP	192.168.2.0/24	anywhere	any	
5	ACCEPT	anywhere	anywhere	any	dpt:tcp 80

Figure 4: An incorrect forwarding chain which allows non-SSH traffic from hosts on the 192.168.2.0/24 network.

example a “witness,” since it illustrates the assertion. Witnesses are less powerful than counterexamples in that the existence of a counterexample demonstrates conclusively that an assertion is false while the existence of a witness only demonstrates that it is possible to satisfy the assertion. Nevertheless, witnesses can be very useful for convincing yourself (or others) that an assertion really holds. They can also be useful for debugging certain kinds of problems that can best be tested with an assertion that you expect to fail.

```
ASSERT EXAMPLE (FROM 192.168.2.*
AND NOT FOR TCP 25)
SUBSET OF DROPPED FORWARD;

Assertion failed.
Counterexample:
TCP packet from 192.168.2.1:6362[eth1]
to 131.106.3.253:25[eth1]
in state NEW with flags[ ].
```

Figure 5: Counterexample for the example assertion.

Suppose that you wanted to ensure that the firewall policy does not block all SMTP connections. To test whether your firewall correctly implements this policy, you might use the assertion:

```
ASSERT EXAMPLE FOR TCP 25
SUBSET OF DROPPED FORWARD;
```

If the policy is correct, the assertion should fail, since the assertion implies that all SMTP packets are dropped. In this situation, a witness can often provide useful information that allows you to discover why the assertion succeeds. Since using assertions in this manner is very counter-intuitive, we provide the user with NOT SUBSET OF and IS NOT operators that can be used in place of the SUBSET OF and IS keywords. This allows the user to avoid using “backward assertions” like the one above. Using these operators, the user can test whether all SMTP packets are dropped as follows:

```
ASSERT EXAMPLE FOR TCP 25
NOT SUBSET OF ACCEPTED DROPPED;
```

This new assertion will hold in exactly the cases in which the old assertion would fail and vice versa, but is much more intuitive to use.

Rule History

Witnesses and counterexamples provide the system administrator with very detailed information about the causes of an assertion failure. Nevertheless, it can be difficult to trace an error to the fault that causes it even when a good counterexample is available. We can obtain more precise information about the particular rules that create an error by constructing a “history map” during generation of the rule set MDD. The history map matches each packet to the set of rules that potentially accept or drop it.

Using the history map, we can associate packets in an assertion’s fail set with a small number of filtering rules – usually much smaller than the number of

rules in the entire policy. This permits the administrator to narrow his inspection of the policy to just a few critical areas. Since the set of rules to examine includes every rule that matches a packet in the assertion’s fail set, it is possible that we may list some correct rules as well as the incorrect ones. In many cases, however, the history map will enable the system administrator to ignore most of rules that are not related to the problem.

Implementation

To evaluate an assertion, we represent each condition using a Multiway Decision Diagram (MDD), a data structure which is suitable for representing and manipulating large sets of packets. An MDD is a directed acyclic graph in which the nodes are organized into levels and all arcs from a node at a given level point to nodes at the level below.

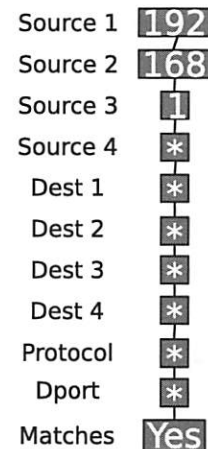


Figure 6: MDD representing FROM 192.168.1.*.

Each level of the MDD corresponds to an attribute such as protocol, connection state, or destination port. For instance, level K , the top level of the graph, represents the first source octet of a packet. The bottom level of the MDD is a special terminal level which indicates whether or not a packet belongs to the match set. For space reasons, our figures show only some of the levels of each MDD. We also use an asterisk as a wildcard character to represent “all arcs not explicitly listed in this node” when many of the arcs leading from a node point to the same child.

To construct an MDD representation of a primitive such as FROM <address> or FOR <port>, we use nodes with all arcs pointing to the same child to mask out all but the relevant levels of the MDD. To represent FROM 192.168.1.*, we start at the top of the MDD and work down, inserting a node with just one arc labeled “192” at the top level (since the top level corresponds to the first octet of the source address). This arc connects to a node at the next level down which has a single arc labeled “168” which points to a node with an arc labeled “1” in the next level. The “2” arc connects to a node representing the fourth source octet

of the condition. All the remaining nodes in the graph are labeled with the wildcard character, since the other criteria are not relevant to the condition. This process is illustrated by Figure 6, which shows a simple condition MDD.

To test whether a particular packet is in the match set of a condition, we simply descend the MDD from its root to a terminal node using properties of the packet to guide the descent. If we reach the “matches” node, the packet is in the set. Otherwise, it is not. In Figure 6, a packet from 192.168.1.1 to 64.130.15.7 on TCP port 25 matches the condition, but a packet from 192.168.4.1 does not, since there is no arc for “4” leaving the node for source octet three.

The primitives ACCEPTED <chain> and DROPPED <chain> require special treatment. To generate MDDs for these conditions, we construct an MDD representation of each firewall chain. We then remove all the paths except those which point to the correct terminal node (either ACCEPTED or DROPPED) using a projection operation.

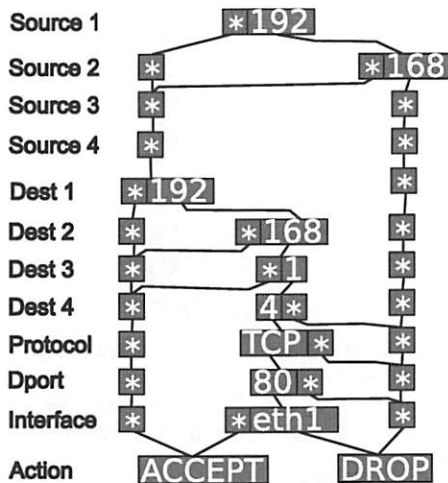


Figure 7: MDD for the FORWARD chain.

Figure 7 shows part of the MDD for the chain in Figure 1. To create an MDD representing ACCEPTED FORWARD, we copy those paths of the chain MDD which lead to the ACCEPTED node into the condition MDD and ignore paths leading to the DROPPED node. The resulting MDD is given in Figure 8.

Complex conditions containing the AND, OR, and NOT operators can be represented by using MDD intersection and union operators to combine the primitive MDDs. An example MDD for a more complex condition is given in Figure 9. Union and intersection can be performed very efficiently using MDDs. Using operation caches, we can obtain a guarantee that each pair of nodes in the graph is visited only once during these operations. Since the number of nodes in the graph is usually much smaller than the number of packets represented by each condition, we can complete these operations very rapidly. The complement

operator is also very efficient. It requires a single descent of the MDD, which is linear with respect to the size of the graph.

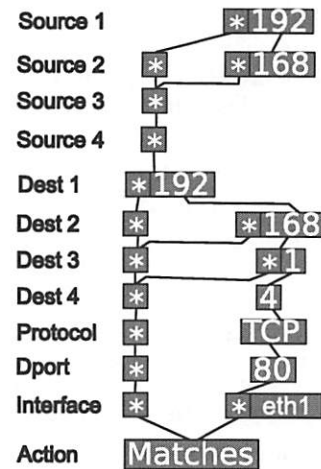


Figure 8: Packets accepted by the FORWARD chain.

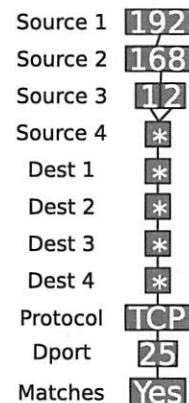


Figure 9: MDD representing FOR TCP 25 AND (FROM 192.168.1.* OR FROM 192.168.2.*).

To determine whether a containment assertion holds, we examine the set of packets that match condition *A*, but do not match condition *B*. If the assertion fails, this set will be non-empty, as illustrated by Figure 10.

The pseudocode in Figure 11 describes this process in detail. First, we construct MDDs representing the packets that match condition *A* and condition *B*, respectively, in steps 1 and 2. In step 3, we use an MDD complement operation to find the set of packets which do not match condition *B*, the right-hand side of the assertion. We intersect the MDD returned by the complement operation with the MDD representing condition *A*, the left-hand side of the assertion, in step 4. This creates an MDD representing the fail set of the assertion. In steps 5 through 8, we test whether the set is empty and return an appropriate value.

To test an equality assertion, we use the algorithm given in Figure 13, which is similar to the

algorithm for testing a containment assertion. Steps 1 through 7 create an MDD representing the fail set. If the fail set is non-empty, we have the situation illustrated by Figure 12 and the assertion fails. If it is empty, the assertion holds.

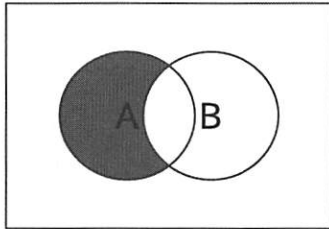


Figure 10: Fail set for the SUBSET OF operator.

```
bool testSubsetAssertion(cond A, cond B):
[1] mddA = condition_to_MDD(A);
[2] mddB = condition_to_MDD(B);
[3] notB = MDD_complement(mddB);
[4] result = MDD_intersect(mddA, notB);
[5] if notEmpty(result) then:
[6]   return ASSERTION_FAILED;
[7] else:
[8]   return ASSERTION_HELD;
```

Figure 11: Checking a containment assertion.

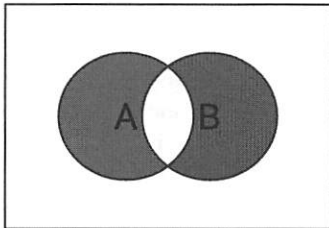


Figure 12: Fail set for the IS operator.

These techniques allow us to determine whether or not a firewall policy satisfies a set of assertions. These operations form a basis for performing more advanced calculations, such as example generation and history analysis.

Implementing Examples

To generate the counterexample for an assertion, we change the algorithms in Figure 11 and Figure 13 to return an arbitrary element from the fail set. This is done by replacing the last four lines of each algorithm with those in Figure 14.

The function *choose_element(X)* picks an arbitrary element from the set represented by MDD *X*. If the assertion does not fail, we choose an element from

the fail set as the counterexample. If the assertion fails, we choose an element from the match set of the left-hand condition as a witness, since the elements of that set must match both conditions. To select an element, the *choose_element* function walks the MDD from the root node to the bottom of the graph, arbitrarily selecting arcs at each level (in practice, we select the first non-zero arc of each node) and storing each selected attribute in a “packet” structure which can be printed at the end of the traversal.

```
bool TestISAssertion(cond A, cond B):
[1] mddA = condition_to_MDD(A);
[2] notB = MDD_complement(mddB);
[3] resultA = MDD_intersect(mddA, notB);
[4] mddB = condition_to_MDD(B);
[5] notA = MDD_complement(mddA);
[6] resultB = MDD_intersect(notA, mddB);
[7] result = MDD_union(resultA, resultB);
[8] if notEmpty(result) then:
[9]   return ASSERTION_FAILED;
[10] else:
[11]   return ASSERTION_HELD;
```

Figure 13: Checking an equality assertion.

```
bool TestSubsetAssertion(cond A, cond B):
[5] if notEmpty(result) then:
[6]   return choose_element(result);
[7] else:
[8]   return choose_element(mddA);
```

Figure 14: Generating an example.

Implementing History

In order to build the history map, we construct a “history MDD” for each built-in chain of the firewall. The history MDD is similar to the MDD for a rule set or an assertion, but has two extra levels at the bottom of the graph. The top levels of the MDD correspond to the levels of a rule set MDD. The extra levels at the bottom represent a chain identifier and an index for each rule. We reserve index 0 for the default policy of a chain and index the remaining rules sequentially starting from 1. Construction of the history MDD is done concurrently with construction of the MDD representation of each firewall chain. As we insert rules into the rule set MDD for the chain, we copy those rules into the history MDD, adding nodes to identify the chain and rule to the bottom levels. If we encounter a rule which matches packets already matched by some other rule, we use MDD union to store a mapping to both rules.

Suppose we want to test the assertion

Chain Forward (Default DROP)					
#	Target	Source	Dest	Flags	
1	DROP	192.168.2.0/22	anywhere		
2	ACCEPT	anywhere	192.168.3.0/24		
3	ACCEPT	anywhere	anywhere	dpt:tcp 25	

Figure 15: Example rule set which protects a network 192.168.4.0/24.

```

ASSERT HISTORY NOT FROM 192.168.2.*
  AND TO 192.168.4.*
  AND FOR TCP 22
SUBSET OF ACCEPTED FORWARD;

```

against the policy given in Figure 15. The assertion verifies that SSH traffic is allowed to a protected network 192.168.4.0/24 unless it originates on an untrusted network 192.168.1.0/24, from which all traffic is blocked by the firewall. The assertion will fail due to a typo in the subnet mask of the source address in rule 1. As a consequence of this fault, an SMTP packet from 192.168.3.1 to 192.168.4.2 that should be accepted will be dropped.

An example history MDD for the rule set is given in Figure 16. To save space, only some levels of the MDD are represented in the figure.

We can use the history MDD to find the rules that match this packet by descending the graph. To make this descent easier to follow, we have highlighted the path representing the example packet in Figure 16. Starting from the root node, we follow the arc labeled 192, since the source address of our example packet begins with 192. We next follow the arc labeled 168.

Because there is a typo in the subnet mask of rule 1, the node we are now examining has arcs for 0, 1, and 3 that point to the same child node as the arc for 2. We follow the arc labeled 3 to a node with all arcs pointing to the same child. We continue past the child node. The destination address of our example packet begins with 192, so we follow the arc labeled 192. We then follow the arc labeled 168 and the arc labeled with a wildcard, which represents all values other than 3.

This brings us to another node with all arcs pointing to the same child, which we continue past. We now take the arc labeled TCP, then the arc labeled 25. This takes us to a node at the chain level. The only arc leaving this node is labeled 1, so we know that the only rules that affect this packet are in the FORWARD chain. Following the arc takes us to a node representing rules 1, 3, and the default policy (represented by

the label 0). Rule 2 is not listed since it matches only packets sent to subnet 192.168.3.0/24.

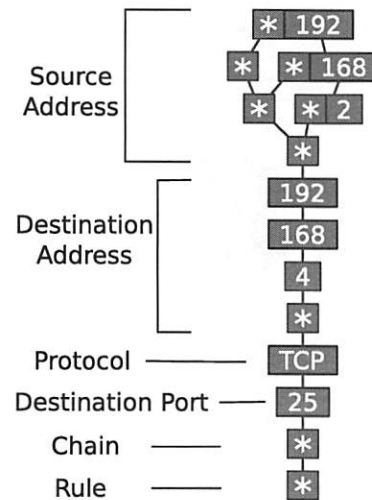


Figure 17: History MDD for an assertion.

This traversal gives the history map for a single packet. To find the rules that match those packets that violate the assertion, we intersect the history MDD for a chain with an MDD representing the fail set of the assertion. The fail set MDD can be computed as for counterexample generation, except that the result must be extended to include levels for the rule index and chain identifier. This can be done by padding the fail set MDD with wildcard nodes at the bottom two levels. This is illustrated by Figure 16, which gives an extended fail set MDD for the assertion.

The top four levels of the MDD correspond to the source addresses of packets that match the assertion. In this case, the assertion matches all packets except those from the 192.168.2.0/24 subnet. The next four levels represent the destination addresses of packets that match the assertion. In the example, only packets to subnet 192.168.4.0/24 match. The next levels represent protocol and destination port. The bottom levels represent the chain identifier and rule index of

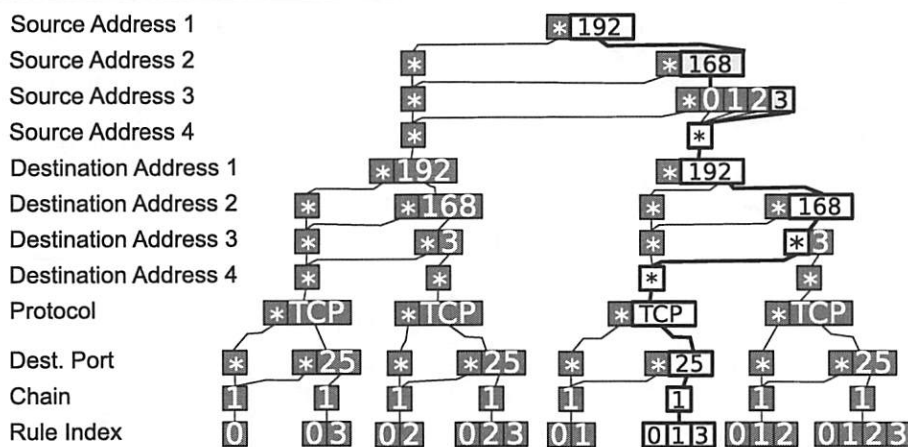


Figure 16: History MDD for a firewall chain.

the packet. Since we have not yet determined which rules are related to the assertion, these are represented as wildcard nodes.

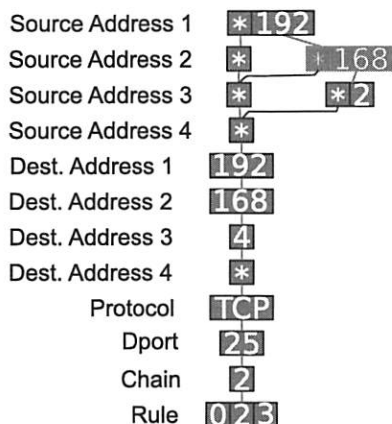


Figure 18: Result MDD after intersection.

Intersecting the extended fail set MDD with the rule set MDD gives us the history map MDD in Figure 16. ITVal converts this graph into the human readable output given in Figure 19. From this output, it is very easy to see that the fault lies in either rule 1 or rule 3. Rule 2 is ignored, since it only matches packets that do not cause the assertion to fail. In a longer policy, other extraneous rules would also be ignored.

```

ASSERT HISTORY NOT FROM 192.168.2.*
AND TO 192.168.4.* AND FOR TCP 25
SUBSET OF ACCEPTED FORWARD;
#Assertion failed.
Critical Rules:
  Firewall 0 Chain 1 Default Policy.
  Firewall 0 Chain 1 Rule 1:
  DROP all -- * * 192.168.2.0/22
    0.0.0.0/0
  Firewall 0 Chain 1 Rule 3:
  ACCEPT tcp -- * * 0.0.0.0/0
    0.0.0.0/0 tcp dpt:25
  
```

Figure 19: Human readable history map.

In this case the fault lies in rule 1. An examination of that rule quickly reveals the typo. Our algorithm also lists rule 3 and the default policy of the FORWARD chain. Rule 3 is the rule that should have accepted the dropped packets and therefore may help the administrator understand the error. The default policy can be ignored, since it always matches every packet seen by the firewall. Using this enumeration of the matching rules, the system administrator can concentrate on the rules directly relevant to the problem.

History and Equivalence Classes

It is often much easier to use assertions than to perform a manual inspection of the policy. For one thing, the rules in a policy interact with each other in ways that can be confusing to the user. One rule in the

policy might mask another rule or cause the rule to be applied only in certain, unusual, circumstances. Because each assertion is independent of the others, writing and understanding a list of assertions is often easier than manually correcting the rule set. More importantly, it is possible to construct a partial or high-level specification of the policy using assertions. This partial specification can ignore many of the details of the policy, which allows it to be simpler than the rule set to which it is applied.

Nevertheless, debugging the firewall using assertions has certain limitations. There is a tradeoff between the completeness of a specification and how easy the specification can be constructed. Deriving assertions that are both useful and effective can be a very challenging task.

Another limitation of the assertion approach is that certain kinds of faults cannot easily be identified using history maps for an assertion.

The policy in Figure 20 is supposed to protect a secure subnet 192.168.2.0/24 from intrusions on an untrusted network 192.168.1.0/24. An assertion checking that legitimate SSH traffic can reach the protected network is also given in the figure. A typo in rule 2 causes the assertion to fail. Unfortunately, the history map for the assertion will show only the default policy. None of the other rules in the policy match any packets in the fail set. In particular, rule 2, which contains the fault, does not match any packets from the 192.168.2.0/24 subnet and, therefore, is not listed.

One way to address this problem is to create an assertion that checks whether packets from 192.168.2.0/24 are accepted. The history map for such an assertion would immediately identify the typo in rule 2. The problem with this is that the system administrator has no way of knowing such an assertion is needed. It is not practical to create assertions for all of the possible typos in a policy, since doing so would require at least as much work as manual inspection of the policy.

A better way to address the problem is to extend the technique described in [9] to provide history information that can be used to discover faults in the policy. In that work, we described a technique for separating the computers on a network into related classes of hosts based on information taken directly from the firewall policy. Hosts in each class are equivalent in the sense that the firewall will accept or drop a packet from (or to) a host in the class only if it will accept or drop an otherwise identical packet from (or to) any other host in the class. For instance, if the firewall drops all SSH packets from host *A*, it will also drop all SSH packets from host *B* if they are in the same class. Each class of hosts on the network is represented as a five level “class MDD” which can be manipulated efficiently using MDD intersection and union operators.

Figure 21 lists the three classes shown in Figure 20. Class 2 corresponds to the untrusted subnet 192.

168.1.0/24. Class 3 is an anomalous class of hosts caused by the typo in rule 2. The existence of this class is an immediate clue to the system administrator that the firewall policy contains a serious error. Class 1 corresponds to all other hosts on the network.

```
QUERY HISTORY CLASSES;
There are 3 total host classes:
Class 1:
    <Everything not in the other classes>
Class 2:
    192.168.1.[0-255]
Class 3:
    192.186.2.[0-255]
```

Figure 21: Equivalence class decomposition of a policy.

Partitioning the hosts on a network into equivalence classes allows us to generate a “policy map” that shows functional groupings of the hosts on a network. The only input necessary to create a policy map is the firewall policy itself. When the policy contains a fault, it will often be manifested in the policy map as a missing class or by the presence of an unexpected class of hosts. The equivalence class technique can detect many kinds of faults that are difficult to identify using assertions. These faults include typos, overly broad rules, shadowed rules, outdated rules, and even missing rules. Unfortunately, while the policy map assists the system administrator in detecting these problems, it provides him with little information that can be used to identify the rules that must be changed to repair the issue.

We can enhance the policy map by annotating each class of hosts with a list of rules that match packets to and from a host in the class. To do this, we extend each class MDD with wildcard nodes. The resulting graph is similar in structure to the condition MDDs used to analyze assertions, but has wildcards at every level except the source address levels. This MDD matches the set of all packets whose source address matches a host in the class. We then repeat the procedure to produce an MDD with wildcards everywhere except the destination address levels. We can now intersect with the history MDDs for each chain to determine which rules match these packets. This intersection generates a result MDD which can be translated into a human-readable history map.

An MDD representing all packets with source address from class 3 is given in Figure 22. The top four levels of the MDD correspond to source addresses on subnet 192.186.2.0/24. The remaining levels contain wildcard nodes.

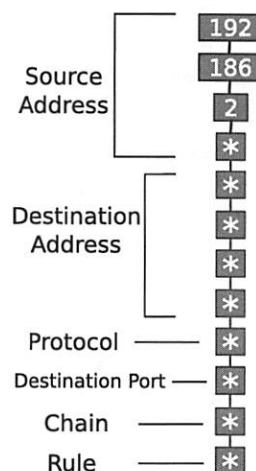


Figure 22: History MDD for class three.

```
Class 3:
Firewall 0 Chain 1 Default Policy.
Firewall 0 Chain 1 Rule 2:
ACCEPT all -- * * 0.0.0.0/0 192.186.2.0/24
tcp dpt:22
```

Figure 23: History Map for class three.

A portion of the history map for the equivalence classes of the policy in Figure 20 is given in Figure 23. The existence of an anomalous class containing hosts from the 192.186.2.0/24 subnet immediately alerts the system administrator to a serious error. A quick glance at the history map for class 3 reveals that only two rules are of interest: the default policy and rule 3. The system administrator now takes a careful look at rule 2 and discovers the fault, which enables her to repair the policy.

Existing Work

There are many good techniques for finding errors in a firewall policy. Tools such as nmap [3], Nessus [5], and fester [1] allow the system administrator to actively test a firewall for specific well-known vulnerabilities. Unfortunately, these tools are

Chain Forward (Default DROP)				
#	Target	Source	Destination	Flags
1	DROP	192.168.1.0/24	anywhere	
2	ACCEPT	anywhere	192.186.2.0/24	tcp dpt:22

```

ASSERT HISTORY TO 192.168.2.* AND
FOR TCP 22 AND
NOT FROM 192.168.1.*
SUBSET OF ACCEPTED FORWARD:

```

Figure 20: A fault that history mapping misses.

little help in identifying the faults which cause an error. For instance, nmap may indicate that a critical network port is unavailable for a variety of reasons: if the host is down, the firewall prohibits access to that port, the TCP wrappers on the server are incorrectly configured, or a routing error makes the host unreachable. Distinguishing between these potential causes is extremely difficult. Once the error has been narrowed down to the firewall, these tools do not provide any information about the policy itself that aid the user in determining why the error has occurred.

More rigorous testing can be done using passive testing tools, such as the Lumeta firewall analysis engine [10, 12], that perform an exhaustive off-line analysis of the policy. These tools simplify the task of determining whether the firewall policy contains errors by allowing the user to specify a set of logical queries that provide information about the policy. Some work has also been done on using expert systems to test the firewall policy [2]. The Lumeta engine provides support for History Mapping and limited example generation. However, the Lumeta engine is a proprietary closed-source product, which does not support iptables. These passive analysis tools also do not provide class-based analysis and therefore require the user to invest a significant amount of time designing appropriate queries or specifications against which the policy must be tested.

Conclusion

Using examples and history mapping, a system administrator can easily identify the two or three critical rules in a rule set that lead to a serious firewall error. Detecting these faults greatly reduces the amount of time an administrator must spend in careful examination of the policy and makes it much easier to manage and maintain a large, restrictive firewall policy. Using counterexamples and witnesses, the system administrator also gains valuable knowledge about the circumstances under which an error occurs. Using rule history with equivalence classes allows the system administrator to quickly and easily detect both errors and faults in the policy without constructing a large number of complicated assertions. The only required input is the policy itself. This greatly simplifies the process of maintaining, debugging, and repairing a restrictive firewall policy on a large network.

The techniques for generating a history map and relevant counterexamples have been implemented in our tool, ITVal, which can be downloaded from <http://itval.sourceforge.net>. The web site also provides several example specification files which can be downloaded and customized for use on a variety of networks.

Author Biographies

Robert Marmorstein is a professor at Longwood University with research interests in system and network

security. When he is not analyzing firewalls, he spends his time avoiding grues in the Great Underground Empire.

Phil Kearns is a Professor of Computer Science at the College of William and Mary. His research interests lie in the general area of computer systems.

Bibliography

- [1] Barisani, Andrea, "Testing Firewalls and IDS With ftester," *Insight, Newsletter of the Internet Security Conference*, Vol. 5, 2001, <http://www.tisc2001.com/newsletters/56.html>.
- [2] Eronen, Pasi and Jukka Zitting, "An Expert System for Analyzing Firewall Rules," *Proceedings of the 6th Nordic Workshop on Secure IT Systems*, 2001.
- [3] Fyodor, "The Art of Port Scanning," *Phrack*, Vol. 7, Num. 51, September, 1997.
- [4] Gouda, Mohamed G. and Alex X. Liu, "Firewall Design: Consistency, Completeness, and Compactness," *Proceedings of the International Conference on Distributed Computing Systems*, IEEE Computer Society, March, 2004.
- [5] Lampe, John, *Nessus 3.0 Advanced User Guide*, October, 2005, <http://www.nessus.org>.
- [6] Marmorstein, Robert, *ITVal Project Website*, 2005, <http://itval.sourceforge.net>.
- [7] Marmorstein, Robert and Phil Kearns, "An Open Source Solution for Testing NAT'd and Nested iptables Firewalls," *19th Large Installation Systems Administration Conference (LISA '05)*, pp. 103-112, December, 2005.
- [8] Marmorstein, Robert and Phil Kearns, "A Tool for Automated iptables Firewall Analysis," *FREENIX Track, 2005 USENIX Annual Technical Conference*, pp. 71-82, April, 2005.
- [9] Marmorstein, Robert and Phil Kearns, "Firewall Analysis With Policy-based Host Classification," *20th Large Installation Systems Administration Conference (LISA '06)*, pp. 41-51, December, 2006.
- [10] Mayer, Alain, Avishai Wool, and Elisha Ziskind, "Fang: A Firewall Analysis Engine," *Proceedings of the IEEE Symposium on Security and Privacy*, May, 2000.
- [11] Stearns, Bill, <http://www.stearns.org/>.
- [12] Wool, Avishai, "Architecting the Lumeta Firewall Analyzer," *Proceedings of the 10th USENIX Security Symposium*, August, 2001.

NetADHICT: A Tool for Understanding Network Traffic

Hajime Inoue – ATC-NY, Ithaca, NY

Dana Jansens, Abdulrahman Hijazi, and Anil Somayaji – Carleton University, Ottawa, Canada

ABSTRACT

Computer and network administrators are often confused or uncertain about the behavior of their networks. Traditional analysis using IP ports, addresses, and protocols are insufficient to understand modern computer networks. Here we describe NetADHICT, a tool for better understanding the behavior of network traffic. The key innovation of NetADHICT is that it can identify and present a hierarchical decomposition of traffic that is based upon the learned structure of both packet headers and payloads. In particular, it decomposes traffic without the use of protocol dissectors or other application-specific knowledge. Through an AJAX-based web interface, NetADHICT allows administrators to see the high-level structure of network traffic, monitor how traffic within that structure changes over time, and analyze the significance of those changes. NetADHICT allows administrators to observe global patterns of behavior and then focus on the specific packets associated with that behavior, acting as a bridge from higher level tools to the lower level ones. From experiments we believe that NetADHICT can assist in the identification of flash crowds, rapidly propagating worms, and P2P applications.

Introduction

Network administrators are regularly confounded by the behavior of the networks they manage. Part of this confusion is a function of the rapid innovation in applications and protocols; it also arises from simple human unpredictability. Much of the blame for the mystery of computer networks, though, can be laid at the feet of our tools: we simply do not have the means for truly understanding what is happening in our networks [11].

To be sure, we have numerous tools for monitoring networks. Packet volume monitors can alert administrators to gross changes in network behavior. Flow reconstruction and protocol dissectors can reveal the behavior of individual connections. Signature scanners can identify specific security problems, and anomaly detectors can tell us that something is “different.” As we discuss in the Related Work section, these tools, while useful, are not sufficient for us to achieve network awareness.

One key piece that is missing is a way to view traffic at the “right” level of abstraction. For example, when dealing with a surge in web traffic, we need more detail than “80% of your traffic is HTTP”; however, analyzing the patterns in 50,000 HTTP connections is sure to induce information overload. In order to address this shortcoming, we need tools that can automatically extract multiple human-comprehensible abstractions of network behavior (e.g., most of connections have slash-dot in the HTTP referrer field). By understanding the structure of the observed abstractions and by seeing how future traffic fits into them, an administrator can quickly come to understand *how* network behavior

changes at a level of granularity that facilitates both holistic understanding and appropriate response.

We have developed a tool called **NetADHICT** (pronounced “net-addict”) for extracting and visualizing context-dependent abstractions of network behavior. NetADHICT hierarchically decomposes network traffic: for example, observed packets are first divided into IP and non-IP groups, IP packets are then split into TCP and non-TCP, and so on. What is notable about NetADHICT, though, is that its decomposition is automatically derived from observed traffic; in other words, it learns an appropriate context-dependent hierarchical classification scheme automatically with *no built-in knowledge* of packet or protocol structure.

NetADHICT is able to perform this feat through the use of a novel clustering method we refer to as “Approximate Divisive Hierarchical Clustering (ADHIC).” When applied to traces of captured traffic, NetADHICT generates hierarchical clusters that closely correspond to the kind of semantically meaningful abstractions that are normally used to describe network behavior [10]. Remarkably, NetADHICT often aggregates packets without the use of ports or IP addresses; it also groups packets across multiple flows when there is significant commonality.

Although we have not attempted to optimize NetADHICT, it is designed to be extremely efficient, allowing use of the tool at wire speed. We have benchmarked NetADHICT at 245 Mb/s. NetADHICT can achieve this speed because it classifies packets using sets of fixed-length strings located at fixed offsets within packets. We refer to these patterns as (p, n) -grams.

Of course, fancy algorithms are not enough to make a useful tool for observing networks; we also

need appropriate interfaces for visualizing and interacting with observed traffic. To that end, this paper describes NetADHICT from the point of view of a network administrator. Specifically, we explain how NetADHICT's interactive AJAX-based web interface can be used to understand network behavior in a new, and we believe highly useful, way – one that gives insights into the sub-protocol behavior of networks in a way that preserves user privacy. Even though NetADHICT is still evolving, it has already proved to be a valuable tool for understanding networks. NetADHICT is made available under the GNU GPL license [13]; we hope that this paper will encourage others to get involved in the evolution of NetADHICT.

The rest of this paper proceeds as follows. The next section explains related work in tools and methods for understanding network traffic. We then give a detailed rationale for the use of hierarchical clustering to analyze networks and subsequently describe the ADHIC clustering algorithm. The interface and implementation of NetADHICT are described in the next section which is followed by a narrative on how one uses the tool in conjunction with other tools to allow administrators to better understand their networks. We then discuss limitations of NetADHICT and plans for future work before concluding by describing how to obtain our tool and a summary of our work.

Related Work

Fundamentally, the problem of understanding network behavior is one of data reduction: we must transform gigabytes of network traffic into a small number of concepts and details that can be understood and acted upon by human administrators. As there are many different ways in which we can choose to understand network behavior, there is correspondingly a variety of tools that are designed to answer different questions.

Network and systems administrators have a variety of mature tools to select from for counting packets and flows. From open source solutions such as MRTG [16] and FlowScan [19] to large commercial offerings such as HP Openview [17] and IBM Tivoli [12], tools are readily available for visualizing the packet and flow statistics records that can be exported by many routers. Such solutions allow an administrator to determine bandwidth consumption on a host, network, or port basis. For well-behaved applications that use standard ports, such views can be useful for identifying deployed applications; as most peer-to-peer protocols and other multimedia protocols do not use standard ports or masquerade as HTTP (port 80) traffic, more sophisticated tools are needed to identify the most bandwidth-hungry applications.

The most common strategy for monitoring evasive applications at the network layer is to combine flow reconstruction with deep packet inspection and application-specific identification rules (signatures). Tools such as Wireshark (formerly Ethereal) [3] are

designed to perform such analysis on captured packet traces. Commercial network forensics tools [4] facilitate ongoing traffic capture so that recent behavior can be queried on demand to provide packet, flow, application, and even user-level views of traffic. In contrast, other commercial products such as those produced by Sandvine [22] can analyze packets at wire speeds, primarily to enable per-application traffic shaping. What these systems have in common is that they rely upon elaborate sets of rules in order to identify, dissect, and even change application behavior. While such rules are generally crafted by hand, there has been extensive research in identifying applications and usage patterns using a variety of machine learning techniques [2, 14, 6].

Network administrators, however, are interested in more than monitoring what applications are running; they also need to know when human intruders or malware have compromised the security of their systems. Intrusion detection systems are most commonly built upon elaborate rule databases that specify the signatures associated with known attacks [21] or specifications of what network activity is and is not allowed [18]. One fundamental problem for all such systems, though, is that disruptive network activity often is caused not by attackers but by highly popular legitimate applications or services (e.g., flash crowds). Either the rules have to be broad enough to not signal alarms in some disruptive conditions, or administrators have to tolerate a regular stream of false alarms.

Anomaly detection systems have the potential to adjust rules to local conditions through the use of machine learning mechanisms; behavior at the network layer, however, is highly dynamic, and in fact attempted attacks are now routine on the Internet. For these reasons, some researchers are coming to believe that anomaly detection at the network layer is a questionable strategy for detecting security incidents [9]; such reasoning, however, does not exclude the use of anomaly detection for detecting networking problems in general [1].

While these various techniques for monitoring and classifying network behavior have their place, there is also a need for tools that can uncover other aspects of the structure of network traffic. The next section explores what is missing in currently available tools.

Understanding Network Traffic

To understand network behavior we need to observe more than changes in packet volume or security status. In addition, we need to discern patterns in traffic that allow us to group related communication activities together. While we cannot hope to find all possible patterns, there are a number of ways to capture simple patterns in network traffic. The general strategy for grouping packets together using automatically

learned patterns (or features) is to employ a clustering-based machine learning algorithm.

One strategy for clustering packets is to learn patterns associated with sets of flows. For example, AutoFocus [7] is a system that hierarchically clusters packets using IP address and port information so that temporal patterns in the activities of groups of hosts can be inferred. However, there are many interesting patterns which cannot be defined in terms of protocol and address. Ma, et al. [14] addressed this limitation by building classifiers from network flows using the header and first 64 bytes of payload from the initial packet in each flow. Their classifiers were built using clusterers. Clusterers don't label data, they simply group it together. Once a clusterer was constructed, the packets in each cluster were examined and assigned to a particular protocol, turning the clusterer into a classifier. Although, Ma's approach is promising, it is not fast enough to be used to monitor networks online. In fact, most general clustering and classification algorithms are simply too slow to learn at wire speed. This limitation is significant because we want to observe changes in behavior as they happen so that we may react to changes in a timely fashion.

NetADHICT differentiates itself from other traffic clustering tools in that NetADHICT clusters traffic without knowing in advance any details about the packets' structure and does so in a way that can execute at the speed of the network. This allows it to find

its own differences between the packets – differences that adapt automatically to the traffic as it changes. Traffic volume patterns can be watched online and compared between the differentiated clusters. These traffic groups are presented visually through a tree: Packets that are found to be internally dissimilar are clustered separately, so that users immediately observe that they are different in nature. Also, each cluster's position in the tree contains a wealth of context about how the traffic for that cluster is similar and different from the rest of the network's traffic. Volume of traffic through each cluster is displayed in the tree in near real-time, allowing administrators to analyze traffic patterns as they emerge.

High level tools can often show that something is wrong. Low level tools give tremendous detail, but often are too slow or too information-rich to be used. Clustering is a form of filtering that allows one to focus on relevant behavior. NetADHICT allows the integration of high and low level tools while providing enough information to allow administrators to move smoothly from high to low.

The NetADHICT interface is designed to facilitate such multi-level exploration. It is built around the cluster tree approach to network volume visualization. Each cluster of traffic within the tree is identified by a traditional classifier and colored appropriately. (The traditional classifier is built around EtherType, IP protocol, and IP port.) This coloring of the clusters, along

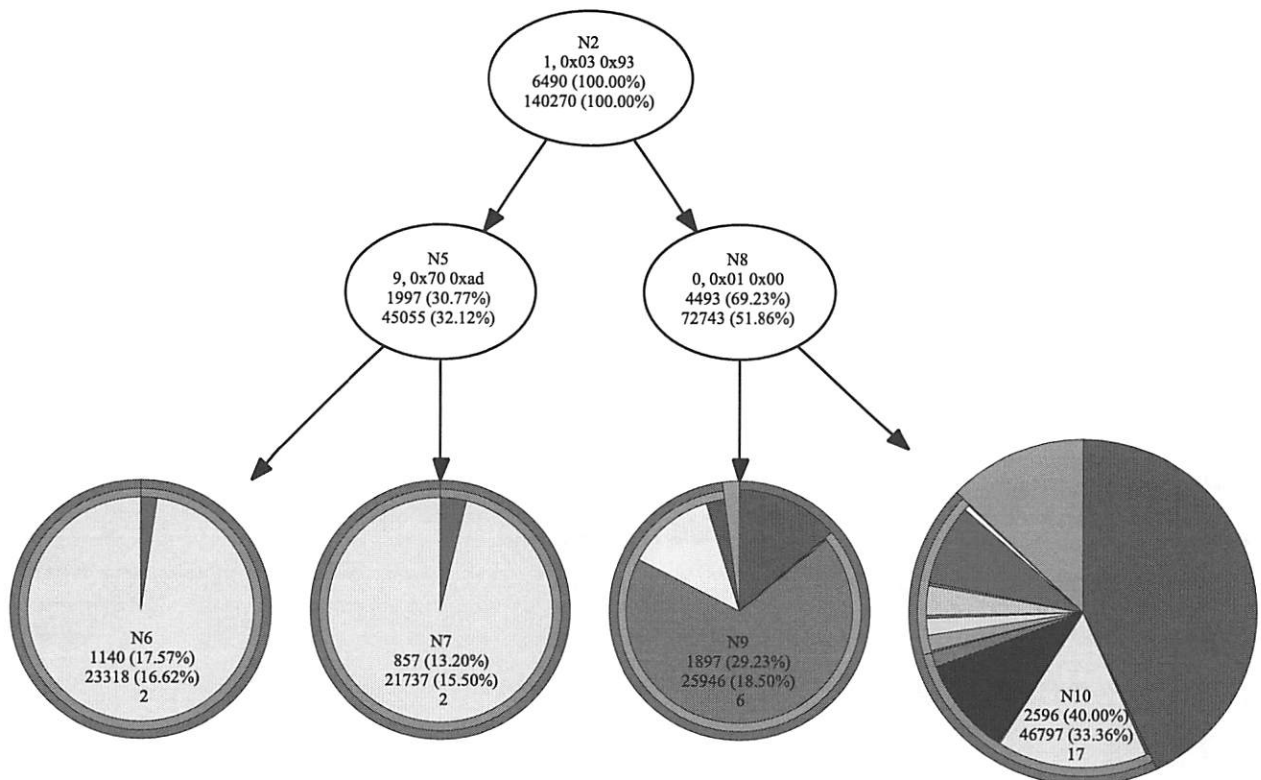


Figure 1: An ADHIC cluster decision tree after 3 node splits. The labelling of the internal nodes and the pie charts of the terminal nodes are explained in the “NetADHICT” section.

with their position in the tree, highlights traffic that is contextually different but using the same ports. It provides an excellent starting point for delving further into different classes of traffic in the network and highlights classes that would have remained completely hidden otherwise. In this fashion NetADHICT is a tool for enhancing network awareness.

Hierarchical Clustering with (p,n) -grams

NetADHICT is centered around a novel clustering algorithm that recursively splits the set of observed packets into smaller and smaller groups. We call our algorithm Approximate Divisive Hierarchical Clustering (ADHIC), the root of NETWORK ADHIC Tool (NetADHICT). This algorithm is classified as a type of machine learning algorithm broadly known as divisive hierarchical clustering [5], but ours is substantially different than other general purpose or network specific clustering algorithms. We give an overview of ADHIC below; for more details, please see Hijazi [10].

The feature we use to split groups of packets is called a (p,n) -gram. (p,n) -grams are used to denote substrings at fixed offsets within packets. The p is the offset and n is the length of the substring. For example, the (p,n) -gram (8, 0xdc3b) denotes the 2 byte hexadecimal substring 0xdc3b 8 bytes into the packet

(these are the middle bytes of the source MAC address in the Ethernet frame).

Divisive hierarchical clusterers form decision trees. The tree is constructed using the **split** and **merge** operations. When the average bandwidth of a terminal cluster (leaf) exceeds a configurable threshold over a certain time window, the cluster is split into two clusters. The existing node becomes an internal decision node and two new terminal clusters are formed. A (p,n) -gram is found by examining the packet cache, a buffer that stores a few percent of recently received packets. (p,n) -gram frequencies are calculated for packets assigned to the terminal cluster in question, and a (p,n) -gram is chosen which matches roughly half of them. Conversely, if a node averages less than a set bandwidth limit, it is deleted and its parent becomes a leaf node. Newly created nodes cannot be merged for a certain number of minutes, called the maturation period, to prevent transient behavior from affecting tree structure.

Through these two operators, ADHIC generates a decision tree that specifies the contents of clusters. The path from the root of the tree to a leaf or other internal cluster specifies the boolean equation of (p,n) -grams which determine if a packet belongs to that cluster.

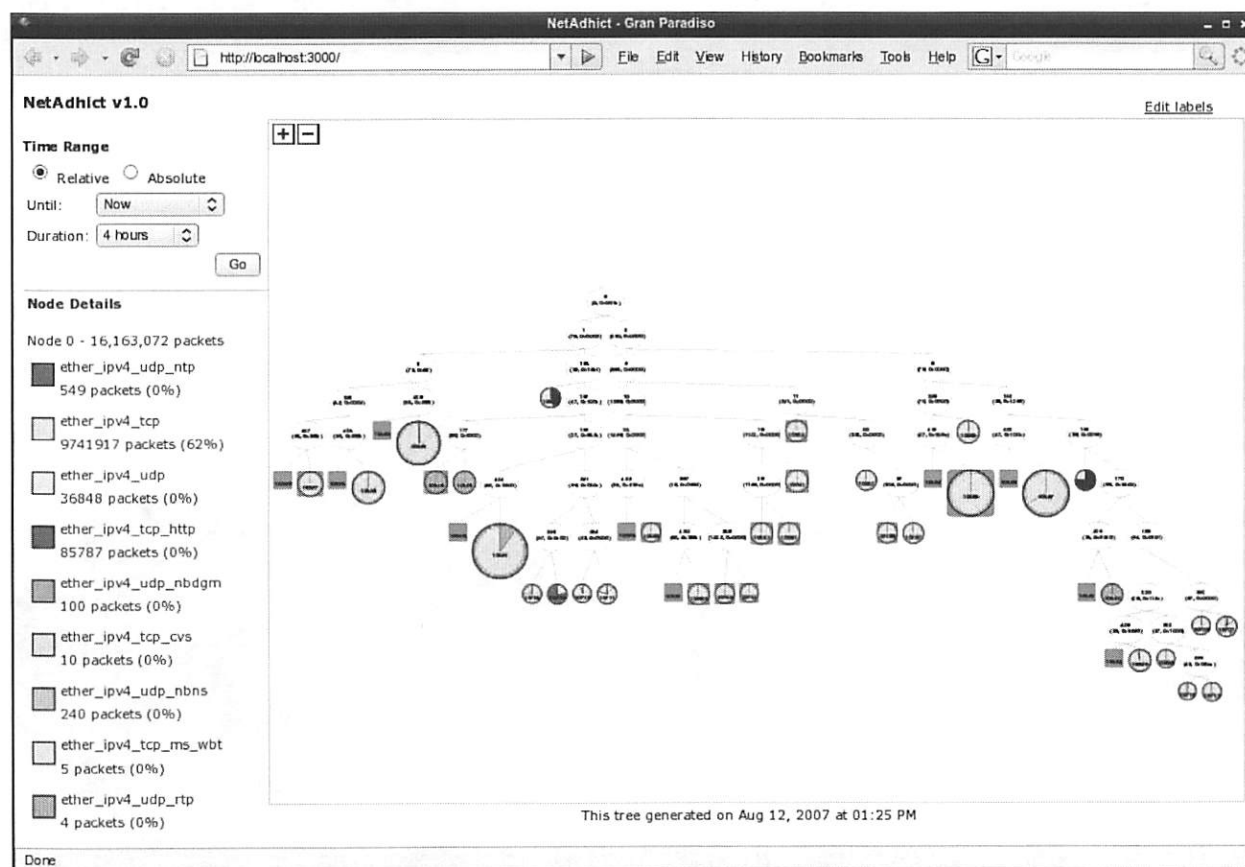


Figure 2: NetADHICT's primary tree view, showing traffic over a four hour period.

Consider, for example, the tree in Figure 1. The root node has a (p, n) -gram of $(1, 0x0393)$ and its child (p, n) -grams $(9, 0x70ad)$ and $(0, 0x0100)$. All the offsets point to locations within the MAC addresses of the Ethernet frame. The offsets 0 and 1 segregate portions of the destination MAC address and 9 distinguishes between source MAC addresses. The left edge signifies that the packet has matched the (p, n) -gram in the parent node. The right edge is followed for packets that do not contain that (p, n) -gram. Please note the labeling of the nodes and the pie charts is not a part of the ADHIC algorithm; it is a visualization produced by NetADHICT and is explained in the “NetADHICT” section.

The decision trees produced by the ADHIC algorithm have a number of strengths as representations of network structure. They are simple in structure and semantics, facilitating user understanding and analysis. Trees can be frozen, or subtrees removed from the learning algorithm, allowing users to directly modify the tree. Subtrees can be incrementally modified and augmented by users in a straightforward way. Additional information and statistics may be easily added to decision trees. Finally, the ADHIC representation also easily lends itself to implementations like the decision tree packet classification algorithms often used as alternatives to TCAM [23].

NetADHICT

NetADHICT’s user interface is an interactive AJAX-based web page (Figure 2). The primary element of this interface is the tree view, which allows the network administrator to quickly see how traffic is being distributed between clusters and what traffic each cluster represents. The tree view can update itself with newly available data as it becomes available, allowing network administrators to watch changes in the traffic’s structure as they occur.

Traffic is shown in the tree view for a selected time period, as shown in Figure 3. The time period is normally selected relative to the present time; it can be set to always show the latest data available or just data that was available in the past. Alternatively, a time period can be specified directly for any time at which there is traffic data available.

In the tree view, internal nodes represent a (p, n) -gram operation which is displayed within each node, such as $(6, 0x0001)$. By tracing a node’s ancestors up the tree, you can see which (p, n) -grams are or are not present in the node’s traffic.

Terminal clusters show traffic volumes for the selected time period, as well as much more information through their size and coloring. The size of each terminal cluster represents the volume of traffic relative to all other terminal clusters in the tree. The sizes make it easy to see where traffic is distributed within the tree from a high level, or within a subtree at a lower level.

The coloring of each terminal cluster represents the basic packet types of its traffic and the traffic’s

labels. Basic packet types include IP, TCP, UDP, and other non-IP traffic. For each cluster, the outer ring’s color shows what percentage of traffic for that cluster, for the specified time period, consisted of IP packets. For IP traffic, an inner ring shows what percentage of the traffic was TCP, UDP, or other IP packets. In Figure 4, TCP traffic dominated but approximately 6% of the traffic assigned to the cluster was UDP packets and another 5% was non-IP packets (ARP in this case).

Time Range

☒ Relative ☐ Absolute

Until: Now

Duration: 30 minutes

10 minutes
20 minutes
30 minutes
40 minutes
50 minutes
1 hour
1.5 hours
2 hours

Go

Figure 3: Selecting a time range for which NetADHICT will display a tree and traffic statistics.

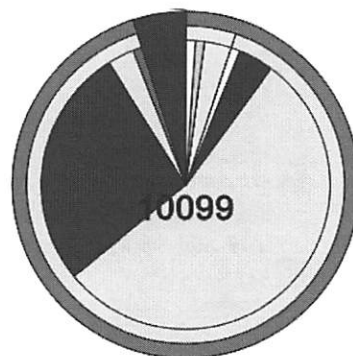


Figure 4: A single cluster in NetADHICT, containing a large number of traffic types.

Labels are used to name and group the semantic classes of traffic for a cluster. NetADHICT provides initial labels and colors with a simple traffic classification by EtherType, IP protocol, and IP port, but a user-defined label can be applied to each classification when a more precise semantic class is determined by the network administrator. The labels for each terminal cluster are represented through the pie charts within the rings. Each color represents a different label and their size represents the percentage of traffic for that cluster which belonged to the given label.

Labels are created, modified, and deleted by the network administrator within the NetADHICT interface. By clicking on the “Edit labels” link in the top right corner, the label editing interface (Figure 5) is displayed. From here the system administrator can

create new labels, remove old ones, or change the names and colors of existing labels.

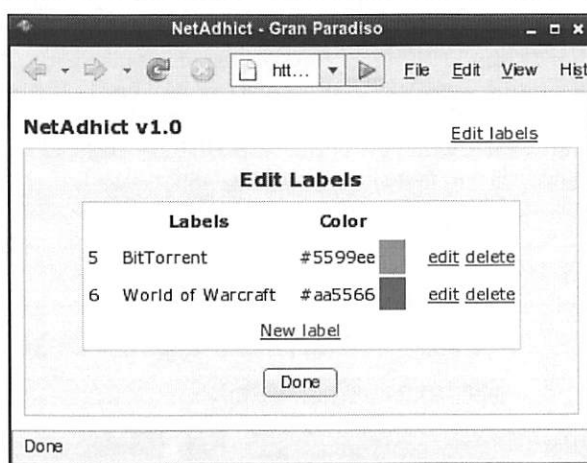


Figure 5: Editing the list of user-defined labels.

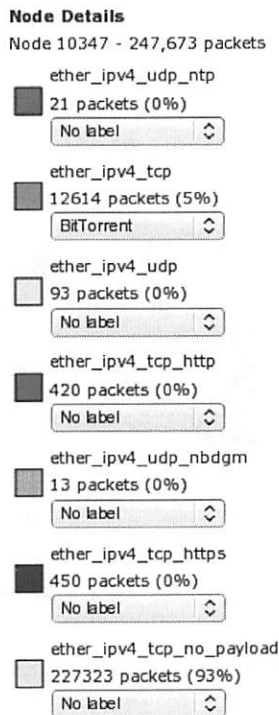


Figure 6: Details of a node in the tree.

Detailed information for any node in the tree, internal nodes and terminal clusters alike, can be displayed by hovering the mouse over the node (Figure 5). The details include the precise volumes of traffic which the node and all nodes below it represent. They also show what traffic classification or label each color in the node's pie chart represents, what percentage of the node's traffic fell into each label, and let you change the labels for the node's traffic.

For more detailed analysis, the traffic represented by any cluster can be exported by NetADHICT into packet dump files. These dumps can then be analyzed

using standard network analysis tools such as Wireshark. Any number of clusters can be selected for a packet capture by shift-clicking on them. While selected for capturing, all packets that match the clusters will be captured to pcap files; these files can be downloaded at any time during the capture.

The NetADHICT backend packet analyzer and frontend web interface both require access to a MySQL database. The frontend is a Ruby on Rails application and so requires a web server configured with support for such applications. A web browser is used to view the NetADHICT interface.

NetADHICT's trees are rendered using standard SVG and JavaScript; however, because many current browsers have poor support for JavaScript-generated SVG, the quality of NetADHICT's interface varies greatly depending upon the web browser used. In fact, at the time of this writing the Firefox 3 alpha release (also known as Gran Paradiso [8]) is the only truly capable browser for running NetADHICT. Firefox 2 is functional but too slow for real use. Internet Explorer 7 and Konqueror 3 lack the SVG support required, and the latest Opera release (version 9.23) contains critical bugs in regard to its SVG support that prevent NetADHICT from functioning. While this variable performance is currently problematic, better SVG support is on the short-term development roadmap for most browsers; thus, we believe the browser compatibility issue will soon cease to be a significant problem for NetADHICT.

Usage Scenarios

We now move from describing the tool itself to how a network administrator would use the tool in several common situations. These situations are:

- 1) checking normal network traffic,
- 2) analyzing a flash crowd,
- 3) recognizing special network usages or activities such as local P2P traffic, and
- 4) identifying and isolating a propagating worm.

In earlier papers [10, 15], we described several experiments which we used to evaluate the ADHIC algorithm. The following descriptions are not of actual experiments, but we draw upon our research experience to describe how one could use the tool in a small network context. While we believe these usage scenarios should generalize to larger networks, further testing is required.

Checking Normal Network Traffic

The first usage scenario is network surveillance under normal conditions. The network administrator begins a session by examining the current tree, as in Figure 2. An administrator knows what a normal-condition tree and flow values (the size of the pie charts) look like and can immediately spot anomalous behavior. If he sees unusual behavior in some part of the tree he can magnify the subtree and examine its volume pie charts to see what is wrong.

Alternatively, he can view the tree for specific past time periods. By viewing the series of trees from the immediate past he can see how the tree developed to its current form. We have found this “movie” to be very useful in understanding network behavior. It is analogous to animated radar maps – but instead it shows network weather.

As we will discuss in the other usage scenarios, anomalous traffic flows will lead to easily identifiable changes in the tree structure or the volume pie charts. However, if the tree visualization is not enough to identify the problem he can hover the cursor over a node to see the statistical network traffic volume summary as in Figure 6.

An administrator has another option if he needs more information than the two visualizations provide. He can shift-click on any number of clusters to request that NetADHICT record all future packets assigned to those clusters. These packets can then be downloaded as a pcap file into a tool like WireShark [3]. This allows the administrator to use any analysis tool he likes that reads pcap files while first using NetADHICT to eliminate the vast majority of uninteresting traffic.

Analyzing a Flash Crowd

Now let us consider what might happen if a particular page on a website was hit by a flash crowd, or “slashdotted.” A traditional network tool would notify the administrator that total traffic volume is up, when the surge occurred, and would probably identify, using port numbers, that HTTP is the culprit. The administrator might then turn to the web server logs for further information.

Like a traditional tool, NetADHICT would show that traffic is up, that it is HTTP, and when the surge in traffic started and if and when it ended. NetADHICT’s other behavior would differ based on how long the flash crowd remains. If it is much shorter than the maturation period, then the flash crowd would be contained in the terminal nodes. These would likely be dominated by HTTP and the administrator would have to refer to the web server’s logs for more information. If the flash crowd remained longer than the maturation period, though, NetADHICT would begin segregating traffic, searching for commonalities in the packets of the flash crowd.

If the flash crowd were from a particular network with a common prefix, NetADHICT might recognize it by specifying the address in the created (p, n) -grams. Or, it might find (p, n) -grams that matched other parts of the header or even payload contents such as the requested URL. The key thing to remember is that NetADHICT will keep refining its classification of incoming packets so long as certain nodes continue to receive more than their “fair share” of packets, and so long as there exists (p, n) -grams that match a significant fraction of a high traffic node’s packets.

The resulting cluster equations (encoded in the structure of the tree) would give the administrator

more information than the traditional tools before he dove into the web logs. Further, because NetADHICT can provide exemplars of the “new” packets, a network administrator would have a significant head start in determining the precise characteristics and origin of the flash crowd.

Identifying P2P Traffic

Peer-to-peer (P2P) traffic is in some ways similar to flash crowds. The differences lie in that many P2P protocols are not well documented and many actively try to evade traditional traffic classification. For example, many BitTorrent clients now provide the option to encrypt their data so that payload inspection cannot identify the protocol [20]. In addition to encrypting its payload, BitTorrent does not use standard ports. Because of this, traditional tools often either cannot classify or misclassify P2P traffic that use these evasion tactics.

NetADHICT does not need to know about protocol structure, so it can be more useful than traditional tools in such situations. In our evaluation experiments [10] we found that ADHIC, when confronted with (unencrypted) BitTorrent as a new protocol, segregated the traffic into just two terminal clusters. These clusters corresponded to the UDP tracker packets and the TCP content packets. Thus, we believe that a network administrator, when investigating a new high network volume application, would see one or a few nodes with increased traffic. If the P2P session lasted several maturation periods, NetADHICT would begin splitting those nodes, building a P2P dominated subtree. Other than the port-based color labeling, NetADHICT cannot classify traffic, but it could provide a pcap file of P2P traffic with all other traffic filtered out for the administrator to investigate.

NetADHICT application-level classifications are more robust than those produced by traditional port-classifying tools. For example, suppose that a P2P client uses port 80 for its traffic. Traditional tools would likely mis-classify the new application traffic as HTTP. In experiments we have found that NetADHICT, because it often ignores ports, continues to cluster P2P traffic correctly even when clients use misleading ports [10].

Identifying Worm Traffic

Finally, let us consider the case of a propagating worm. Worms are a subcase of the flash crowd because they involve a large increase in traffic over protocols that are already used. The benefits of NetADHICT are similar to those during a “slashdotting.” If a particular set of addresses is responsible, NetADHICT may allow the administrator to discover it by knowing the (p, n) -gram offsets. Similarly, NetADHICT might also create new clusters specific to worm traffic. Inspection of the (p, n) -gram equation might allow the administrator to construct a signature for the worm, which could then be applied to a firewall or an intrusion prevention system. Even if the

(p, n) -gram equations could not be used for this purpose, the pcap file NetADHICT provides would be useful in signature construction.

We believe NetADHICT would be useful in many additional scenarios. These, however, hopefully provide a useful introduction in how the tool could be used. NetADHICT, as a complement to existing tools and systems, can allow an administrator to quickly ascertain the state of his network and investigate any anomalous behavior as it occurs.

Discussion

While our experience with NetADHICT has convinced us that it is a useful tool for network administrators, we have also experienced its limitations. First and foremost is ADHIC's inability to classify packets. ADHIC is a clustering algorithm that does not rely on prior knowledge to segregate traffic into clusters – therefore the clusters may not always have the semantic splits that administrators may prefer. ADHIC's use of (p, n) -grams also is a source of problems. Some interesting semantic splits do not lend themselves to using (p, n) -grams. The reason for this is that some identifiers are not at constant offsets from the beginning of the packet. These weaknesses are mitigated by NetADHICT's incorporation of a traditional classifier, its ability to work with other network analysis tools using pcap files, and the administrator's ability to label the cluster tree.

The extent to which these inherent limitations will affect NetADHICT's usefulness is not clear. To this point we have not investigated NetADHICT's behavior beyond our own laboratory's network. We have determined that NetADHICT can often segregate encrypted and multimedia traffic, by only using their packet headers. If traffic moves to more evasive strategies such as overloading common protocol ports, playing with the other header fields, and using encrypted payloads, NetADHICT may be less able to discover patterns in network traffic.

Evaluation of NetADHICT is difficult because it uses full packet payloads. Using full payloads introduces privacy concerns. This has not been an issue for our internal lab, but we could not investigate NetADHICT's behavior on other, larger, networks because of our inability to correlate the generated trees with the raw pcap traces.

Our limited ability to evaluate NetADHICT has helped motivate us to release this tool. We can advance our research and improve it with more administrators testing NetADHICT on their own networks.

The tool continues to improve, even without an external user base. We are improving the integrated traditional classifier. In addition, we would like to give users the ability to control the tree through manipulating or locking nodes as it grows. This may improve an administrator's ability to find, label, and track

semantic classes of traffic. We are also investigating different user interfaces and visualizations of the data.

Finally, we are working on moving the cluster and sampling portions of NetADHICT into the Linux kernel. This would greatly improve efficiency and increase throughput. Also, the kernel implementation allows us to schedule or filter at a per cluster level, allowing us to use NetADHICT to actively manage network traffic to improve resource allocation and mitigate malicious activity [15]. We see such extensions as a fruitful area for future research.

Availability

NetADHICT is licensed under the GNU General Public Licence (GPL), version 2 or greater. It is available for download at the CCSL software website at <http://www.ccsl.carleton.ca/software>.

Conclusion

NetADHICT shows great promise in aiding administrators in understanding network behavior. It provides a new way to separate traffic that is normal from “interesting” traffic that an administrator is interested in analyzing. With new ways to visualize traffic and providing a network “weather” map, NetADHICT allows administrators to see the status of their whole network at a glance, while also providing ways to investigate smaller flows of traffic. While development and testing are ongoing, by acting as a bridge from higher level analysis tools to low level ones, NetADHICT has the potential to improve how administrators manage network resources.

Acknowledgments

This work was supported by the Discovery grant program of Canada's National Sciences and Engineering Research Council (NSERC) and MITACS.

About the Authors

Hajime Inoue is a Principal Scientist at ATC-NY in Ithaca, NY. He was previously a Postdoctoral Fellow at the Carleton Computer Security Laboratory. He received a B.S. in biophysics from the University of Michigan in 1997 and a Ph.D. in Computer Science from the University of New Mexico in 2005. He can be reached at hinoue@ccsl.carleton.ca.

Dana Jansens is currently studying towards a B.C.S. at Carleton University specializing in networks and operating systems and works as a research assistant in the Carleton Computer Security Lab. In her spare time, she leads the Openbox window manager project and studies social justice issues. You can reach her at dana@ccsl.carleton.ca.

Abdulrahman Hijazi graduated with his masters degree in CS at Johns Hopkins University in 2003 with highest honors. He previously had seven years of work experience as a system analyst/programmer. He

is currently pursuing his Ph.D. in computer science at Carleton University. You can reach him at ahijazi@ccsl.carleton.ca.

Anil Somayaji is an assistant professor in the School of Computer Science at Carleton University and is associate director of the Carleton Computer Security Laboratory. His research interests include computer security, operating systems, complex adaptive systems, and artificial life. He received a B.S. in Mathematics from the Massachusetts Institute of Technology in 1994 and a Ph.D. in Computer Science from the University of New Mexico in 2002. He can be reached at soma@ccsl.carleton.ca.

Bibliography

- [1] Barford, Paul, Jeffery Kline, David Plonka, and Amos Ron, "A Signal Analysis of Network Traffic Anomalies," *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, pp. 71-82, ACM Press, New York, NY, USA, 2002.
- [2] Bernaille, Laurent, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian, "Traffic Classification on the Fly," *SIGCOMM Comput. Commun. Rev.*, Vol. 36, Num. 2, pp. 23-26, 2006.
- [3] Combs, Gerald, et al., Wireshark, 2007, <http://www.wireshark.org>.
- [4] Corey, V., C. Peterman, S. Shearin, M. S. Greenberg, and J. Van Bokkelen, "Network Forensics Analysis," *IEEE Internet Computing*, Vol. 6, Num. 6, pp. 60-66, Nov/Dec, 2002.
- [5] Duda, R. O., P. E. Hart, and D. G. Stork, "Pattern Classification," *Unsupervised Learning and Clustering*, pp. 517-599, Wiley, Second Edition, 2001.
- [6] Erman, J., A. Mahanti, and M. Arlitt, "Traffic Classification Using Clustering Algorithms," *Proceedings of the ACM SIGCOMM Workshop on Mining Network Data (MineNet)*, 2006.
- [7] Estan, C., S. Savage, and G. Varghese, "Automatically Inferring Patterns of Resource Consumption in Network Traffic," *Proceedings of ACM SIGCOMM*, 2003.
- [8] Mozilla Foundation, *Firefox 3 alpha: Gran paradiso*, <http://www.mozilla.org/projects/granparadiso/>.
- [9] Gates, Carrie, and Carol Taylor, "Challenging the Anomaly Detection Paradigm: A Provocative Discussion," *Proceedings of the 2006 Workshop on New Security Paradigms*, ACM Press, 2006.
- [10] Hijazi, Abdulrahman, Hajime Inoue, Ashraf Matrawy, P. C. van Oorschot, and Anil Somayaji, "Towards Understanding Network Traffic Through Whole Packet Analysis," Technical Report TR-07-06, School of Computer Science, Carleton University, 2007.
- [11] Hughes, Evan, and Anil Somayaji, "Towards Network Awareness," *Proceedings of the 19th Large Installation System Administration Conference (LISA '05)*, pp. 113-124, USENIX Association, 2005.
- [12] IBM, *Tivoli software*, <http://www-306.ibm.com/software/tivoli>.
- [13] Jansens, Dana, Hajime Inoue, and Abdulrahman Hijazi, *Netadhict*, <http://www.ccs1.carleton.ca/software>.
- [14] Ma, J., K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker, "Unexpected Means of Protocol Inference," *Proceedings of ACM Internet Measurements Conference*, 2006.
- [15] Matrawy, A., P. C. van Oorschot, and A. Somayaji, "Mitigating Network Denial-of-Service Through Diversity-Based Traffic Management," *Applied Cryptography and Network Security (ACNS'05)*, pp. 104-121, Springer Science+Business Media, 2005.
- [16] Oetiker, Tobias, "Mrtg - The Multi Router Traffic Grapher," *Proceedings of the 12th Systems Administration Conference (LISA '98)*, USENIX Association, 1998.
- [17] Hewlett Packard, *Management Software: HP Openview for Large Business*, <http://h20229.www2.hp.com>.
- [18] Paxson, Vern, "Bro: A System for Detecting Network Intruders in Real-Time," *Proceedings of the 7th USENIX Security Symposium*, USENIX Association, 1998.
- [19] Plonka, Dave, "Flowscan: A Network Traffic Flow Reporting and Visualization Tool," *Proceedings of the 14th Systems Administration Conference (LISA '00)*, pp. 305-317, USENIX Association, 2000.
- [20] *Azureus Project*, <http://azureus.sourceforge.net/>.
- [21] Roesch, Martin, Snort - Lightweight Intrusion Detection for Networks, *Proceedings of LISA '99: 13th Systems Administration Conference*, pp. 229-238, USENIX Association, 1999.
- [22] Sandvine, Inc., *Sandvine: Intelligent Broadband Networks*, 2007, <http://www.sandvine.com>.
- [23] Taylor, David E., "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, Vol. 37, Num. 3, pp. 238-275, 2005.

CAMP: A Common API for Measuring Performance

Mark Gabel and Michael Haungs – California Polytechnic State University, San Luis Obispo

ABSTRACT

Accurate performance testing of heterogeneous distributed systems, such as those created using GRID technology, requires a consistent method for retrieving system performance data from multiple platforms. This paper presents CAMP¹: a low-level platform independent performance data API designed for use with distributed testing frameworks.

CAMP is not necessarily tied to the distributed testing task: it provides a simple, low-level interface into operating system performance data that can be used to build complex performance measurement applications. This paper discusses CAMP's functionality and implementation in detail. It also contains a detailed analysis of the API's correctness, performance, and overhead.

Introduction

Performance testing is a critical part of the distributed system development cycle, and there is a clear need for robust, automated, and reusable testing mechanisms. This task is made difficult by several factors, and each must be individually addressed and resolved for a system to be generally applicable to an appreciable portion of the set of distributed systems.

Test beds often consist of heterogeneous systems composed of different platforms and capabilities. For purpose-built systems, this can be intentional. However, for some system developers, this type of test bed may be the only resource available. For GRID-based systems, this is inevitable.

With the increasing popularity of Java and other virtual machine or interpreter driven languages, application code is often made "incidentally portable;" that is, the final system does not necessarily need cross-platform capabilities, but it gains that ability through the features of the host language. This provides a unique opportunity for distributed system developers to test code on expanded sets of test beds, including the previously mentioned heterogeneous systems. For developers to exploit this advantage, a performance testing framework must be able to operate seamlessly and transparently on several platforms, and it must not provide a burden to developers who are developing otherwise platform-independent code.

Aside from platform independence, a testing framework should be able to measure data that is both relevant and meaningful. Unfortunately, distributed metrics are difficult to generalize. For example, developers of distributed agent systems might be concerned with end-to-end processing time of requests, while a developer of a distributed computation system might

be interested in the efficient and total utilization of available network resources. However, certain metrics do lend themselves to standardization: system performance counters on individual nodes. A platform-independent method of accessing performance data on individual nodes could serve to supplement or even build larger, domain-specific metrics.

The source of system performance data varies. Some solutions make use of highly accurate hardware performance counters [21], which are dependent on the underlying architecture of the tested systems. In addition, the interfaces to these hardware systems usually require extension of the host operating system, making the framework doubly dependent on both the operating system and the underlying architecture. While this provides very accurate data for specific systems on a system-wide level, it is difficult to extrapolate isolated data for a single process.

At a higher level, modern operating systems provide interfaces to performance data, both on the system-wide and per-process levels. These interfaces can take the form of a well-defined API, a high-level system of performance counting, or simply exposed kernel structures that contain performance information. Operating system performance interfaces often make use of hardware performance counters as well, further strengthening their accuracy. In addition, several operating system-dependent metrics are of use to the distributed system developer. These can include virtual memory usage, page fault counting, and network interface statistics.

This paper introduces CAMP: a cross-platform low-level API for measuring system performance, designed for use with distributed testing frameworks. CAMP is grounded on a single proposition: modern operating systems function similarly, and they keep track of their performance data in some externally-accessible way. CAMP standardizes access to this data through a cross-platform interface, fully encapsulating the available operating system performance reporting

¹Mark Gabel work performed his portion of this work at California Polytechnic. Current affiliation: Center for Software Systems Research, University of California, Davis.

services. It is implemented across three major platforms in Python, a platform-independent interpreted language. The API and its respective semantics are identical across each platform. While not a testing framework in itself, CAMP makes a valuable contribution to the task by solving the system data independence problem at the lowest level.

CAMP provides a common entry point and retrieval format for system-wide and per-process performance data. It is implemented completely statically, holding no persistent state. It attempts to provide raw, unprocessed data wherever possible. The implementation is substantial, production-ready, and uses the highest-performing, lowest-overhead method available on each platform to provide data. CAMP provides novel solutions to several implementation-specific problems. These include addressing a process consistently across multiple operating systems, encapsulating time-averaged data in a single, stateless function call, and addressing locally-named devices in a platform independent manner.

Possible Usage Scenarios

CAMP is a versatile interface: it aims to be the foundation of a performance measurement system and can be used as is or easily extended. It is usable in a variety of different scenarios.

Derived Functions

The data provided by CAMP is raw in nature; that is, it has no meaningful calculation performed on it. For example, network connections are measured by total numbers of packets sent and received rather than a rate. CAMP's ability to provide this low-level data across multiple platforms allows developers to create platform independent secondary functions.

For example, a developer needing rate or throughput data for network connections is free to implement the function in any way: her or she can choose how the state is stored, what statistical functions will be used to determine the rate, and how often the system will be polled. The developer is guaranteed reliable and consistent raw data from the CAMP interface. This extension would not be tied to a specific platform. By selectively determining what extra processing and state

are used, the developer minimizes the overhead of the performance measuring system. A sample rate function appears in Figure 1.

System Monitoring

CAMP could also be used in a wide scale client-server monitoring system. The API's versatility and consistency would allow implementation of a variety of cross-platform monitoring systems. System analysts might be interested in passive performance logging systems, where CAMP's data could be used to make decisions about expanding capacity or more efficient allocation of resources. System administrators could find derived functionality like active throughput and capacity meters useful: spikes or drops in measured data could quickly indicate problems.

Using CAMP all of these systems could be built with little knowledge of the inner workings of the tested systems' kernels. The simplistic model and implementation in a clear, scripting-like language allows non-developers to intuitively collect desired performance data.

Distributed System Testing

Accurate operating system performance data would be a valuable addition to an existing distributed testing framework, and CAMP would be able to provide this functionality with minimal overhead. Many existing testing frameworks rely on the distribution of timestamped, domain-specific performance data [7, 11, 18]. In this scenario, timestamped data from CAMP or CAMP-derived functions could be collected along with the existing data and correlated by time. This could provide powerful insight for the diagnosis of performance problems observed on a global level: developers could precisely quantify their software's effect on various operating system services during periods of poor performance.

Other systems concern themselves with treating distributed systems as black boxes [1]. These systems test only externally measurable values like global latency and throughput of provided services. CAMP provides a low overhead, non-intrusive way of supplementing this domain-specific data with system performance data. The API allows developers to maintain this black box model by utilizing the available operating

```

01 # Calculate the current transfer rate (Both
02 # in and out) for the given network adapter.
03 def BulkByteTransferRate(adapter):
04
05     # Query CAMP for the initial state.
06     initial = GetNetBytesSent(adapter)
07     initial += GetNetBytesRecv(adapter)
08
09     time.sleep(SAMPLE_INTERVAL)
10
11     # Query CAMP for the final state.
12     final = GetNetBytesSent(adapter)
13     final += GetNetBytesRecv(adapter)
14
15     return (final - initial)/SAMPLE_INTERVAL

```

Figure 1: A derived function that measures network throughput.

systems to probe processes externally rather than instrumenting the running code.

Design and Architecture

The architecture of a performance test can vary across different project domains, and it can evolve as a single project grows over time. Because of this, the architecture must be flexible; that is, it must be applicable to any of the aforementioned scenarios. To accomplish this, the interface must be created at the lowest level possible. Figure 2 shows the CAMP API in relation to the surrounding implementation levels.

CAMP's design uses a simple, low-level stateless interface. To the user, all performance data can be thought of as existing in a set of permanent counters, and the API merely provides simple accessors to the data. The API is a set of solid, well-tested building blocks that form the foundation of a platform-independent performance monitoring application.

The design is largely inspired by declarative languages like SQL. The data is assumed to exist in a

changing but directly inaccessible state, and the methods for accessing the data are thread-safe and consistent for a single point in time. CAMP focuses on providing the simplest, most intuitive interface without

Performance measuring applications, testing frameworks			
Possible CAMP extensions: derived functions, stateful counters			
CAMP			
Low-level, stateless performance functions			
Linux(proc)	Win32(pdh)	Solaris(kstat)	Other

Figure 2: The CAMP system architecture.

prematurely optimizing it by forcing it into the mold that best suits the performance data access patterns of a particular platform. As discussed in our empirical evaluation, this does take a toll on performance in some cases. The simple, atomic request-response architecture allows a focus on correctness and usability; CAMP defines distinct function contracts, and its proper use is

Global Functions	
GetPercentProcessorTime(cpu)	Global CPU usage. CPU parameter may be omitted.
GetFreePhysicalMemory()	Global free physical memory
Network Functions	
GetNetBytesSent(interface)	Total bytes sent on interface
GetNetPacketsSent(interface)	Total packets sent on interface
GetNetBytesRecv(interface)	Total bytes received on interface
GetNetPacketsRecv(interface)	Total packets received on interface
Disk IO	
GetNumReads_Disk(disk)	Number of read operations on the given disk
GetNumWrites_Disk(disk)	Number of write operations on the given disk
GetNumReads_Partition(partition)	Number of read operations on the given partition
GetNumWrites_Partition(partition)	Number of write operations on the given partition
Per-process Functions	
GetNumPageFaults(process)	Number of major and minor page faults
GetCPUTime_Total(process)	Process CPU utilization
GetCPUTime_User(process)	Process user mode CPU utilization
GetCPUTime_Kernel(process)	Process privileged mode CPU utilization
GetWorkingSetKb(process)	Size of the process working set in KB
GetVMSizeKb(process)	Size of the used virtual address space in KB
GetThreadCount(process)	Number of threads contained in this process
Enumeration Functions	
EnumDiskPartitions()	Enumerates the available disk partitions
EnumPhysicalDisks()	Enumerates the available physical disks
EnumNetworkInterfaces()	Enumerates the valid inputs to the network functions
GetCPUCount()	Returns the number of CPUs in the system
GetProcessIdentifier(pid)	Returns a "process identifier" for the given pid
GetProcessIdentifiers(name)	Returns a "process identifier" for each running process launched from an executable of the given name
<i>All CPU-time functions have an optional second parameter: the sample interval.</i>	

Figure 3: The current API.

not defined by a set of unnecessarily complex access patterns.

Cross-Platform Coverage

For CAMP to be complete, it should be able to provide substantial coverage of each implemented platform's performance statistics. This presents several challenges: some platforms provide more information than others, while some platforms simply provide different information or different levels of granularity.

CAMP makes use of a lowest common denominator design. This approach, described in [5], is taken by the Java Abstract Windowing Toolkit (AWT) [22]. The Java AWT takes the intersection of natively-available GUI components on several platforms and generalizes them under a single interface. While effective, this method does limit the functionality to that of the least functional platform. However, it is guaranteed to be fully consistent in any implementation.

While this intersection design has the potential to severely limit functionality, CAMP is still able to provide a comprehensive API. As discussed previously, CAMP is based on the postulation that modern operating systems function similarly and record similar data. While this data may be difficult to access, the common set of accessible data was more than enough to provide a usable API. The current API appears in Figure 3.

Performance

CAMP is designed to be "as fast as possible," and this involves the elimination of some design alternatives – namely, the ability to leverage preexisting native utilities to shorten CAMP's development time. Most² of CAMP's functionality can be collected from the output of a native, command line driven utility on the host. However, this is a potentially costly layer of indirection: the forking of a process to satisfy a single function call, which may be called several times per second, puts an unnecessary strain on operating system resources and is very time consuming.

This overhead may be acceptable for simple system monitoring tasks, as demonstrated by Eddie [12], but this level of resource consumption is not satisfactory for a high-performance API that intends to form a foundation for performance measuring applications. Instead, CAMP makes use of the fastest, most direct native interfaces into the kernel performance data for each implemented platform. It does not make use of any other utilities or services.

Implementation

CAMP's public interface is implemented in the Python language, and its internals consist of a mix of Python and native code. It currently supports the Win32 (Windows NT/2000/XP), Linux 2.6, and Solaris (SunOS kernel 5.x) platforms. The Win32 implementation interfaces to Microsoft's performance data handler

(PDH) interface. In the Linux implementation, CAMP interfaces with the `proc` filesystem. On Solaris, CAMP uses both the `proc` filesystem and the kernel statistics chain (`kstat`).

This chapter discusses each platform's implementation in detail, and it also chronicles several global issues that affected all platforms.

Python

Python is an open source, interpreted language that is available for most modern platforms. It is characterized by its unusually clear syntax and strong set of libraries. Python lacks end-of-statement delimiters, forced declaration of variables, and explicit types. It also uses the concept of "significant white space," with indentation actually indicating the nesting level of a particular statement. These features lend to Python code's readability. Python combines the simplistic syntax and excellent text processing capabilities of a scripting language with the robustness and maturity of a full, heavyweight programming language.

CAMP uses Python for a number of reasons. Python has a fully wrapped interface to the Win32 interface using the `pywin32` package. This included a wrapper around Microsoft's performance data handler interface, which was able to provide nearly all performance measuring functionality in the Win32 implementation. Python is also ideal for text processing tasks. Containing a full regular expression matching and replacement implementation, Python lessened the difficulty of parsing the cryptic output of the `proc` filesystem.

Windows

Microsoft does provide a low-level interface to performance information: the `HKEY_PERFORMANCE_DATA` registry branch. However, it is not visible to the user and requires direct programmatic interaction with the registry hive. When using the registry directly, access is not error-checked or thread safe. Incorrect use may provide false data rather than an exception or error. Instead of accessing this data directly, Microsoft recommends the use of the performance data handler interface.

The Win32 performance data handler interface is a high-level encapsulation around the concept of performance data gathering. It revolves around the concept of a "counter," which is an object that is attached to particular performance "concept" and can be called to periodically gather data about that concept. To retrieve data, the user calls either a raw or formatted data retrieval function on the counter. The Windows "Performance" control panel administrative tool demonstrates the direct use of this interface.

Windows provides counters for performance "objects." Examples of objects include "Memory," "Processor," "Paging File," and "Physical Disk." Each object is associated with one or more "counters." For

²But definitely not all.

example, the “Memory” performance object contains around twenty counters, including instances such as “Pages/sec,” “Available Bytes,” and “% Committed Bytes in Use.”

Some counters are associated with “formatted” data, which involves a computation on raw data. “Pages/sec” is an example of one of these: the counter itself contains a raw count of the number of pages written to and read from disk, but retrieving the counter value causes an average rate to be calculated. One can bypass this calculation by retrieving the “raw” performance data from a counter. Other, scalar counters like “Available Bytes” return the same value for both formatted and raw outputs. CAMP almost exclusively uses the raw data, which may have its ultimate source in the NT kernel (in the case of process information) or in an individual device driver (in the case of network or disk IO).

A counter can optionally be associated with an “instance.” For the counters in the object “Process,” the set of instances consists of the set of running processes. For the counter “Network Interface,” the set of instances consists of the set of installed network adapters. Microsoft provides a function to enumerate the set of instances for a given counter, which CAMP uses to implement each of the enumeration functions.

The Microsoft performance data handler interface is designed for higher-level usage than that provided by CAMP, but there appears to be little overhead in using its clear, relatively simple interface.

Linux

On the Linux platform, CAMP collects operating system performance data through the `proc` pseudo-filesystem. `proc` is a file-like interface to kernel data structures that show system information, which includes performance data. At the top level, the filesystem contains three types of entries: status files, kernel-specific directories, and process directories³. The `proc` filesystem is referred to as a pseudo-filesystem because the “files” presented are actually file-like interfaces into kernel data structures which reside completely in memory or are generated dynamically. As a result, access to the filesystem is exceptionally fast. Several files are redundant: human-readable versions are supplemented by simple, one-line white space delimited versions that can be parsed more quickly.

The status files contain useful information about the system configuration: information about the CPU, mounted filesystem, attached devices, and memory. It also contains an accurate uptime count, which lists the current system uptime and how much of that time has been spent in the idle process.

The kernel-specific directories are groupings of status files. For example, the `net` directory contains

³Note that this breaks from the traditional UNIX model of only keeping process information in `proc`. This greatly aided CAMP’s development on Linux.

information about each protocol in use as well as raw performance data for each network adapter. In addition, when granted root-level access, several of these files are writable. Modifying one of these “files” causes the modification of the related kernel data structure.

CAMP uses these kernel-specific files for implementing all global functions. In many cases, the data provided by `proc` is already in a sufficient format. In some cases, however, the data requires manipulation. For example, most memory measurements in Linux are stored as page counts. CAMP converts these to a raw size by adjusting the value based on the results of the `POSIX getpagesize` call.

The process directories contain useful information about each running process. They are named by the process identification number (`pid`) of the related process. A process directory contains a symbolic link to the executable that spawned the running process, information about memory and CPU usage, a list of open file descriptors, and varied information about the process’s environment.

The information provided by `proc` fit well into the CAMP interface. It is relatively low-level, accurate, and can be accessed with little overhead.

Solaris

Performance data on Solaris comes from two sources: the `proc` filesystem and the kernel statistics chain (`kstat`). The Solaris `proc` filesystem differs greatly from the Linux implementation, following a more standard UNIX pattern. It exports performance data *only* for processes, not system-wide statistics, and the files contain binary data that must be read within a C-compatible language and cast to the appropriate struct type. This effectively precluded direct access from Python. To solve this problem, CAMP uses its own C to Python shared library that handles all data retrieval, marshaling, and error checking. Once again, reads on the `proc` files are atomic and unbuffered to prevent data inconsistency.

Global system data comes from a large data structure within the SunOS kernel called the `kstat` chain. The data structure consists of a linked list of performance counters, each of which can be one of several types: a set of name/value pairs, a binary C structure, or an undefined “raw” type. The data is categorized hierarchically by “class,” “module,” instance number, and name, but this organization is not reflected structurally; that is, no matter what the search criteria, finding an appropriate `kstat` instance is always an $O(n)$ operation.

A research-quality implementation of a Python to `kstat` bridge already existed [2], but it was not compatible with the current version of the SunOS kernel and it was incapable of retrieving many of the P “`kstat`” instances that CAMP needed. Building on this system, CAMP generalized the solution to all instances

and introduced compatibility with modern kernels. CAMP's version of the library also includes a faster search algorithm: the original author made several passes of the chain when one would suffice. This library also includes several bug and data type conversion fixes, and it also adds the ability to produce enumerations of all statistics structures of a certain class or module.

With these enhancements, all global and enumeration functions were able to be implemented with a relatively simple interface. As in the Linux implementation, memory data in page units had to be converted to a byte count.

Implementation Issues

This section presents selected implementation issues and their respective solutions.

Raw Data on Windows

The pywin32 package that encapsulated the Windows performance data handler interface was incomplete. The existence of the "raw" retrieval function was not implemented in the Python wrapper DLL, and its existence was not acknowledged or documented.

CAMP includes an extended version of the pywin32 interface that includes this missing functionality. CAMP uses Microsoft's documentation and header files as a strict contract for converting the raw native values into appropriately typed Python objects, taking special care to avoid narrowing conversions that may cause loss of data.

Determining CPU Usage

Determining CPU usage from a single function call posed a unique problem. At any instant in time, the usage of a CPU in a time-shared operating system is either zero or one hundred percent. Introducing a persistent polling thread into the system was not a viable option; it would violate the design goals of the interface.

To solve this problem, the CPU functions query the current idle and uptime counters immediately, block for a specified interval, and query a second time and return. This causes no additional performance overhead: the task can be unscheduled while blocked, and there is no thread-spawning overhead. The sample rate is configurable at call time by applying an optional second parameter to the call indicating the block interval. This value is set at 100 ms by default, and has proved to be very accurate while keeping the function responsive.

The utilization is calculated as:

$$\left(1 - \frac{\text{idle}_{\text{final}} - \text{idle}_{\text{initial}}}{\text{uptime}_{\text{final}} - \text{uptime}_{\text{initial}}}\right) \times 100$$

or, more simply, the percentage of time not spent in the idle process. The per-process CPU usage functions are implemented similarly but with a finer granularity, as both kernel time and user time are reported separately.

Another issue arose with multi-CPU systems. CAMP should provide the ability to distinguish load on

a single CPU from global system load, so the global CPU measuring function allows an optional CPU parameter: a 0-based index that requests the load for a specific CPU. Omission of the parameter causes the total system load to be returned. CAMP also provides a CPU count function to provide information when the count is not known.

Enumeration Functions

The networking and disk functions must operate on a per-interface and per-disk or partition level to be useful. However, device names are certainly not consistent across systems, and indexes, while consistent, are arbitrary and not meaningful.

One solution would be to use common device names to reference the devices. The logical candidate for this would be to map Linux's standard eth* hd/sd naming scheme into Windows. However, this would provide an unnecessary burden and layer of confusion for Windows developers. In addition, the mapping of English device names to a universal "code" could prove to be nondeterministic, forcing developers to use empirical tests to determine what "code" referenced the relevant network adapter or disk.

CAMP's solution is to use enumeration functions. In this particular context, the function enumerates the list of installed network adapters and the list of installed disks. However, the semantics of the function is stronger than it appears on the surface. Every output of the enumeration function is guaranteed to be a valid input to the performance measurement functions; this means that the output will not only contain the correct name, but it will also be in the correct format required by the underlying platform.

The Windows build of CAMP uses Microsoft's PDH object enumeration function. On Linux, this was implemented by parsing a file in the proc filesystem and building a tuple of the results. On Solaris, this data is gathered over one traversal of the kstat chain.

Process Addressing

The standard method of addressing a function in an operating system is the process identifier, or pid. However, Windows does not have an $O(1)$ method for programmatically accessing a process's performance counter based on its pid; one must enumerate all processes and look for a match. This would be an unreasonable solution for CAMP: because the API does not maintain a state, each call to a per-process function would take $O(n)$ on a Windows machine (where n is the number of currently running processes).

On Linux and Solaris, the opposite is true. Getting process information by pid is achievable in $O(1)$ time, but finding a set of pids from a name is an $O(n)$ operation.

CAMP's solution to this is to introduce an abstract concept: the "Process Identifier." The developer can retrieve a process identifier through a CAMP function

that takes a pid as its input. Because this function ideally only runs once per session of use, the $O(n)$ running time is acceptable.

On the Windows side, this function is implemented by enumerating all current processes and finding the matching process. CAMP then creates a process identifier, which, on this platform is a string, according to Microsoft's naming conventions. The identifier consists of the name of the executable binary with its extension truncated, and an increasing numeral appended for processes spawned from the same binary. CAMP returns an error value if the process is not found.

The Linux and Solaris functions are much simpler: they merely attempt to open the process's status file from its proc directory. If it fails for any reason (existence of the process or lack of permissions), an error value is returned.

CAMP does not attempt to hide the value of a process identifier in an abstract class. If a developer is aware of the identifier type for the current platform, he or she can bypass the process identifier step.

For convenience, CAMP also provides a function that returns a list of process identifiers given an executable file name. This function returns a list because multiple processes can be running from the same executable – which may often be the case in a distributed system. For example, the prefork variant of the Apache web server creates multiple processes to handle child requests. A developer can simply run `GetProcessIdentifiers('httpd')` to get an enumerable list of all running Apache processes.

Result Validation

The objective of the result validation phase was to ensure that each performance function reports reasonable and predictable behavior under the presence of controlled conditions. CAMP's test plan involves running the performance monitoring functions against small programs that use a specific resource in a measurable amount.

Correct implementations of these programs would require kernel augmentation on all three platforms, as the operating system is inherently in control of the

distribution of resources. In spite of this, our programs were able to provide a reasonable approximation for CPU, memory, network, and thread usage.

The test bed consisted of a high-end PC⁴ running VMWare workstation. Each test ran sequentially on three virtual machines: Microsoft Windows 2000 Professional (kernel 5.0), SuSe Linux 9.3 (kernel 2.6.11), and Sun Solaris 10 (kernel 5.10). Each operating system ran only its essential services to minimize interference with tests. The virtual machine configuration limited each to 256 MB of RAM and a 32-bit virtual processor.

CPU Time Verification

To test the various processor time functions, CAMP was set to monitor a simple Python program that generates a fixed load on a single CPU. Put simply, this program divides a discrete interval into an "on time" and an "off time" based on the desired load, and alternates between busywaiting during the "on time" and sleeping during the "off time." The code for this function appears in Figure 4.

```
01 def waste(amount):
02     on_time = _INTERVAL*(amount/100.0)
03     off_time = _INTERVAL-on_time
04     ctime = clock()
05     while True:
06         if clock() - ctime >= on_time:
07             sleep(off_time)
08             ctime = clock()
```

Figure 4: The CPU load generation code.

The global CPU time function and the per-process equivalents were first individually tested against idle, 50% and full system loads. Next, both functions were tested against two processes, each consuming approximately 30% of the CPU. The optional "sample rate" parameter was omitted, leaving a default sampling period of 100 ms.

All reported values are averages of 200 samples taken at a 0.5 second interval. Each average is the center of a 95% confidence interval on the mean, given that the overall sample distribution is normal. The algorithm used is described in [10]. The range of each

⁴The host PC has a 64-bit Athlon processor, 2 GB of main memory, and two 10,000 RPM SATA disks.

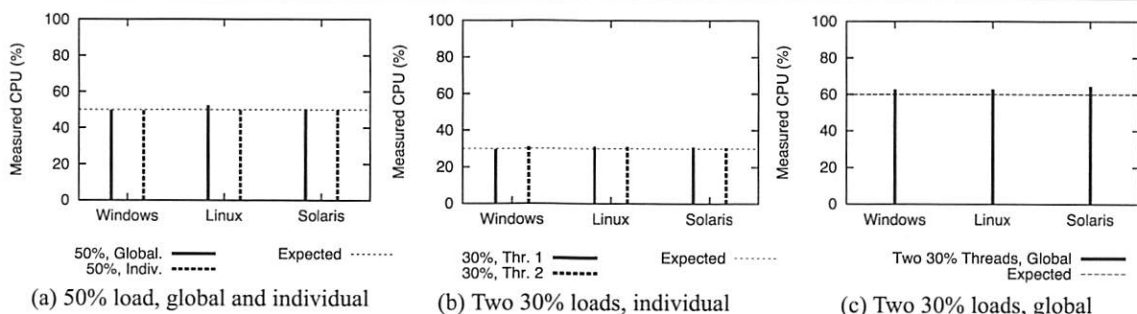


Figure 5: CPU time verification results.

confidence interval never exceeded 0.5% and does not cause a visible difference on the scaled graphs.

The results of these tests on appear in Figure 5. All platforms reported the idle and full loads without error, and each platform was able to provide accurate results for all other intermediate loads. The global measurement of the two-process test read slightly above the expected value; this is most likely due to scheduling overhead.

Memory Usage Verification

The memory usage tests operated on the functions `GetWorkingSetKb` and `GetFreePhysicalMemory`. The test program is a simple, two-stage Java program. It first launches and immediately allocates a 976 KB array on the heap and waits. Upon continuing, it allocates another 976 KB array and waits until terminated. Table 1 lists the results of the two CAMP functions before and after the allocation on all three platforms.

CAMP reported consistent working set results for the program on all three platforms, albeit with a 4KB disparity on Linux. The Linux virtual machine's page size is 4KB, so this represents a reasonable difference of just a single page.

However, while the working set function reported consistent results, it did not report the 976 KB allocation precisely. Because Java programs run in a virtual machine that handles allocations on behalf of the running code, precise changes in memory usage, from a program perspective, are not necessarily exactly reflected at the physical level. In practice, however, the difference between the expected and actual values was just over 1% when using these short-lived test programs. This was sufficient for verification.

The free physical memory validation was somewhat more informal. A precise test for this function is impossible to execute in user space as the full working sets of all tested kernels seldom converge on a completely steady state. This is due in part to each operating system dynamically controlling paging and various caches through periodically-running background

threads. Instead of a precise test, this measurement was a verification that this function reasonably reflected the expected changes in free physical pages. In all cases, CAMP's free memory function reported a value within 10% of the expected result.

Network Utilization Verification

CAMP's network functions report the cumulative bytes and packets sent and received on a single interface. To verify these values, the functions were run against a Java program that sent and received a fixed number of packets and terminated. Each test packet consisted of a plain, addressed UDP packet with a 32 byte payload. UDP traffic was ideal for this test because it allowed a precise prediction of the number of packets sent and received; there was no extraneous ACK or connection setup overhead to skew the results. Testing occurred on a private network consisting of only the CAMPTested virtual machines and their host.

Table 2 shows the results of this test. All platforms reported identical packet counts, but Windows reported a different bytes received count. The offending byte count was 600,000, which amounts to 60 bytes per packet. All other byte counts were 740,000, or 74 bytes per packet, which gives a difference of 14 bytes per packet. Several more tests with increased packet sizes showed that this discrepancy always remained at exactly 14 bytes per packet.

The composition of each 74 byte packet is likely 14 bytes of Ethernet header, 28 bytes of UDP overhead, and the 32 bytes of payload. A likely explanation for the 14 byte difference is that the network driver for the virtual machine's network adapter is not adding the Ethernet header to its incoming byte counter. CAMP's network functions reflect this discrepancy because the byte counts come directly from the respective network drivers. Informal tests on non-virtual Windows machines did not show this same asymmetry.

Despite this difference, the key metrics for evaluation were the packet count and the verification that the byte counts were incremented by at least 320,000

	Before Alloc.	After Alloc.	Difference
Work. Set Size	9132	10120	988
Phys. Free	416244	415272	(972)
(a) Windows			
	Before Alloc.	After Alloc.	Difference
Work. Set Size	14128	15112	984
Phys. Free	10196	9324	(872)
(b) Linux			
	Before Alloc.	After Alloc.	Difference
Work. Set Size	13096	14084	988
Phys. Free	387440	386472	(968)
(c) Solaris			

Table 1: Memory allocation results.

bytes ($\text{payload} \times \text{packetcount}$). In addition, the test programs verified that the payload was delivered in full.

Thread Count Correctness

Verifying the thread count function was a straightforward task. The function `GetThreadCount` recorded the current thread count of a Java program running one extra thread. After a fixed amount of time, the program spawned a second thread. The thread count was recorded once again. The results of this test appear in Table 3. CAMP recorded the proper increase in thread count on each platform.

	Before launch	After launch	Difference
Windows	9	10	1
Linux	9	10	1
Solaris	9	10	1

Table 3: Thread count test results.

This test could have been potentially incorrect. The Java Language Specification does not guarantee that each conceptual thread is backed with a native thread. However, in practice, the Sun-provided HotSpot virtual machine implementation behaves in this manner on all three platforms. Surprisingly, each platform showed the exact same thread count – which would suggest that the virtual machines are structurally similar.

Informal Verification

The previous sections contain strictly quantified test results that confirmed CAMP's functionality. However, they were not comprehensive – they had no reference to the enumeration functions, the disk IO functions, or the page fault count. Due to the difficulty in generating predictable results, these functions were part of an informal test plan.

The disk IO functions are difficult to test consistently. The disk IO data provided by CAMP comes directly from the respective disk drivers, so this data does not reflect an operating system-level layer of abstraction. There is no direct relationship between a user-level read or write and an actual physical disk action. For this reason, the disk functions were verified informally. On all platforms, the counters were integral, increasing, and strictly monotonic. They also increased their respective rates of increase during periods of high disk activity, making them suitable for a derived rate function.

The page fault function was also difficult to test. Instead of a direct analysis, CAMP's output was verified to match that of a comparable native utility on each platform.

On Windows, Microsoft provides an optional utility in the Windows Resource Kit called `pstat`, which provides a crude version of UNIX `ps`. This utility was able to provide a page fault count for comparison. On Linux, the standard `ps` command is able to provide a page fault count with the switch `-O min_flt,maj_flt`. Sun does not provide a utility with Solaris 10 to access the page fault count of a single process. However, a comparable utility, `pio` [9], was able to provide the needed data to verify CAMP's function.

The values returned by the enumeration functions are by definition platform and system-dependent, making them impossible to test quantitatively. Instead, the functions were tested informally on an expanded test bed, which included a 64-bit Linux production server, two quad-processor 64-bit Solaris servers, a Windows XP x64 desktop, and the three original virtual machines.

Testing the enumeration functions involved manually collecting a list of required results in a platform-

	Before Send	After Send	Difference
Packets Sent	37412	47412	10000
Bytes Sent	2175911	2915911	740000
Packets Received	58299	68299	10000
Bytes Received	13001375	13601375	600000
(a) Windows			
	Before Send	After Send	Difference
Packets Sent	60968	70968	10000
Bytes Sent	4157951	4897951	740000
Packets Received	112454	122454	10000
Bytes Received	154652907	155392907	740000
(b) Linux			
	Before Send	After Send	Difference
Packets Sent	1762	11762	10000
Bytes Sent	160145	900145	740000
Packets Received	3379	13379	10000
Bytes Received	386700	1126700	740000
(c) Solaris			

Table 2: Network test results.

dependent manner. On all six test systems, the CPU count was known beforehand and simple to verify. On Windows, correct values for the network, disk, and partition enumeration functions are available in the “Device Manager” administrative tool.

On Solaris and Linux, the command `netstat -i` provided a correct list of network interfaces suitable for comparison. Disk and partition lists were available as symbolic links in the device node directories. This information was available in `/dev/disk/by-id/` and `/dev/dsk` on Linux and Solaris, respectively.

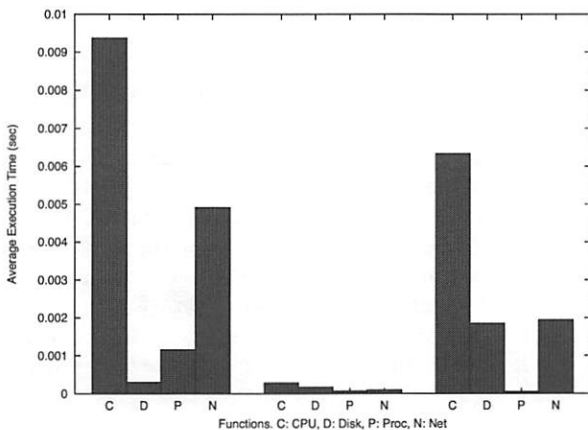


Figure 6: Execution time results. Functions are grouped by similar implementations and averaged. From left to right: Windows, Linux, and Solaris.

Performance Evaluation

This section details the tests used to measure the API’s execution speed and overhead. The first set of tests precisely measured execution speed and CPU usage distribution throughout the implementation components. The second test measured the overhead of a simple CAMP-based monitoring program on a running system. The final test measured the impact of the same monitoring program on a production web server using externally-derived statistics.

Timing and Profiling

This section contains a low-level, direct measurement of CAMP’s performance overhead. Python provides a timing package, `timeit`, which takes a small snippet of code as an input and aggregates several iterations of it into a single execution unit. `timeit` then runs that larger unit several times, timing it using the highest resolution clock available on the host platform.

Timing runs consisted of a total of 30,000 runs per function: three separate execution runs of 10,000 invocations. To measure the overhead of the CPU functions in their purest form, the sample rate was set to zero to eliminate the normal blocking.

For brevity, each “execution unit” comprised groups of similarly implemented functions rather than individual calls. For a function to be “similarly implemented,” it must access the same class of performance

data. A trial run on all individual functions confirmed that the grouped functions had near-identical execution times.

Figure 6 displays the results of these timing runs. All CAMP functions executed quickly, with the slowest function still allowing for over 100 invocations per second.

The Linux functions all executed at an order of magnitude faster than the Solaris or Windows counterparts. The Windows functions all displayed relatively significant overhead, but the disk counters were surprisingly fast to access. The Solaris functions behaved as expected: The disk and network functions both executed in around the same amount of time as they both traverse the same chain for their target data, while the CPU functions must traverse the chain twice and perform some calculations. Solaris per-process functions make use of the `proc` filesystem, so execution time is comparable to that of Linux.

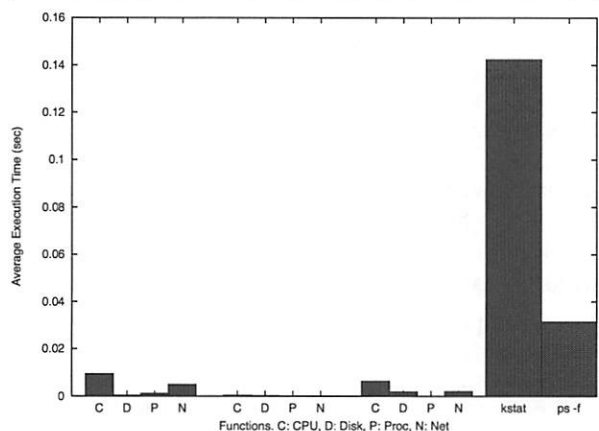


Figure 7: Relative execution time results.

To put these execution times in perspective, Figure 7 contains the same information as Figure 6, but the graph has been scaled against the execution time of running a native process within Python. As discussed previously, this is an approach taken by other utilities and was an implementation alternative for CAMP. On Linux, the timing includes forking a second process, executing the command `ps -f`, and collecting its output. On Solaris, the timing performed a similar action using the `kstat` command line utility. No such command line utility is built in to Windows.⁵ CAMP’s native access of performance data clearly allows a much higher level of performance.

Measured Overhead, or CAMP on CAMP

Next, we used CAMP to measure the overhead of a CAMP-derived monitoring program.

The first task involved a tunable monitoring program, `probe`. This program calls nine CAMP functions

⁵The aforementioned `ps` tool from the Microsoft Resource Kit is parameterless, which forces the full enumeration of every process and *all* of its performance attributes. Its execution time is on the order of seconds.

at a time, with the entire nine-function block being executed at a specific real-time sample interval. To implement the sampling frequency as accurately as possible, this monitoring program times the execution of the nine-function block at startup and adjusts the sleep interval accordingly.

The tests consisted of running probe at increasing sample rates and measuring the impact on the system at each level using another instance of probe. While the results of every CAMP function were included in the measurements, the only significant differences manifested themselves in the global and per-process CPU functions. However, this exercise did confirm that CAMP is indeed stateless and without memory or thread leaks on all implemented platforms. The results of this test on the three implemented platforms is shown in Figure 8. CPU values are an average of 20 samples taken over a 10 second period.

These results were in line with the values recorded by the timing functions. On Solaris and Windows, CAMP produces little overhead up to 10 function blocks/second (90 functions/second). On Linux, CAMP produces little overhead at all measured sample rates.

Impact on an Operational System

The final performance test consisted of measuring CAMP's overhead from an externally accessible statistic. This test is important for several reasons. First, the results of the previous test could have been partially skewed by using CAMP to test itself – caches could have been kept artificially hot. Second, the concept of “CPU utilization” is a somewhat coarse statistic that does not always directly map to actual or perceived performance. Third, CAMP may cause performance im-

pacts in the kernels of the respective platforms in a non-obvious way that may skew results. Lastly, it allows a complete distrust of the operating system-provided performance data.

For this experiment, the target application was a web server. Each virtual machine was configured with Apache 2.0.53. The Linux and UNIX machines used the prefork multiprocessing module (MPM), and the Windows machine used the default Windows NT MPM. Each Apache installation used the default MPM configuration parameters, including initial process and thread count.

The test involved running the probe program from the previous test at the same sample intervals, but instead of recording system performance data on the server, Hewlett Packard `httpperf` [13] collected http client response statistics from an external machine.

To generate a normal load, `httpperf` created a total of 1400 connections to each web server at a rate of 40 connections/second. This rate produced an approximate 15% load on the Linux and Solaris machines and a 30% load on the Windows server. The results of this test are presented in Figure 9.

CAMP only produced a measurable impact on the Windows server. It appeared to take a progressively increasing hit as probe's sample rate increased. CAMP did not affect the performance of the Solaris and Linux servers at this sample rate. This was somewhat unexpected, given that CAMP's performance on Solaris was comparable to its Windows performance. A likely explanation lies in the implementation of the Apache MPM on each platform. The Windows version of Apache handles all client connections from within a single, multi-hundred-threaded process. In terms of

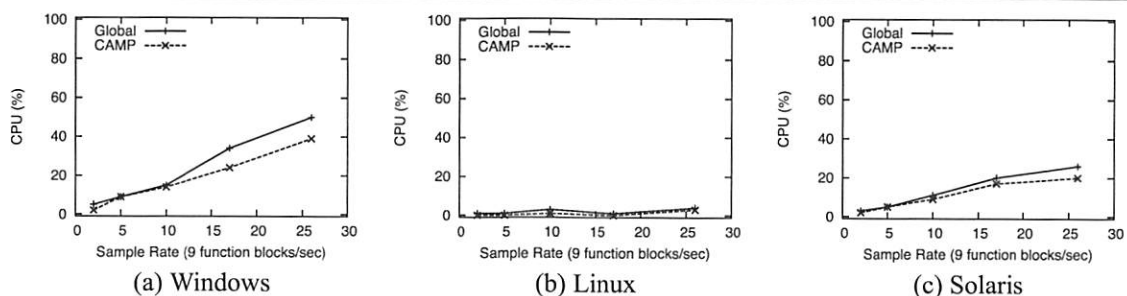


Figure 8: Operational impact at increasing sample rates.

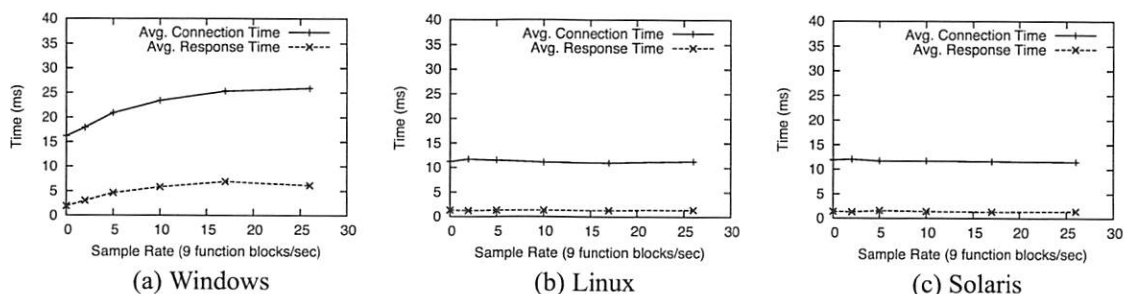


Figure 9: Results of the operational system impact test.

process scheduling, this lone process has the same weight and priority as probe. This causes CAMP's impact to be more pronounced, even at relatively low sample rates. On Solaris and Linux, Apache prefork responds to the increased load by forking additional processes, which causes the Apache system as a whole to be scheduled more often than probe.

Use Cases

CAMPmon

CAMPmon is a simple Python application that demonstrates the use of the CAMP API. It monitors a selected group of networked workstations, each CAMP-supported, and records the global CPU and memory usage to a server console application.

The workstation client program is simple: it merely loads Python's lightweight XML-RPC server and registers the relevant CAMP functions as network-accessible. The server console program reads in an XML file with a list of accessible workstations and queries them at a specified quantum using the Python threading package. This data is then aggregated on the server console and displayed as formatted text.

Testing was performed across eight workstations: four booted into Fedora Linux and four booted into Windows XP. The server monitoring application was launched from a Windows workstation. The system

performed as expected, continuously and accurately reporting each workstation's performance information.

This simple implementation is intended to be a proof-of-concept display of CAMP's potential. Only a single client application was used for both platforms, and the server was unaware of the underlying implementation on each workstation.

A Two-Platform Evaluation of Apache

Two identical servers were configured with Windows 2000 Server and SUSE Linux 9.3 (kernel version 2.6), respectively. Apache 2.0.53 was loaded on each. The Windows server was configured to use the default NT MPM with 300 threads, and the Linux server was configured with the worker multi-processing module (MPM), also configured with 300 initial threads.

A Python script was written to monitor each server. At a half-second interval, it wrote a log file containing network traffic counts and aggregate process resource usage data about the collective Apache processes. This script was ran individually on both platforms.

A third server was configured with a build of Apache flood. Each server was tested with an identical configuration: 250 clients, connecting in groups of 10 at one second intervals. Figure 10 contains the results collected by CAMP.

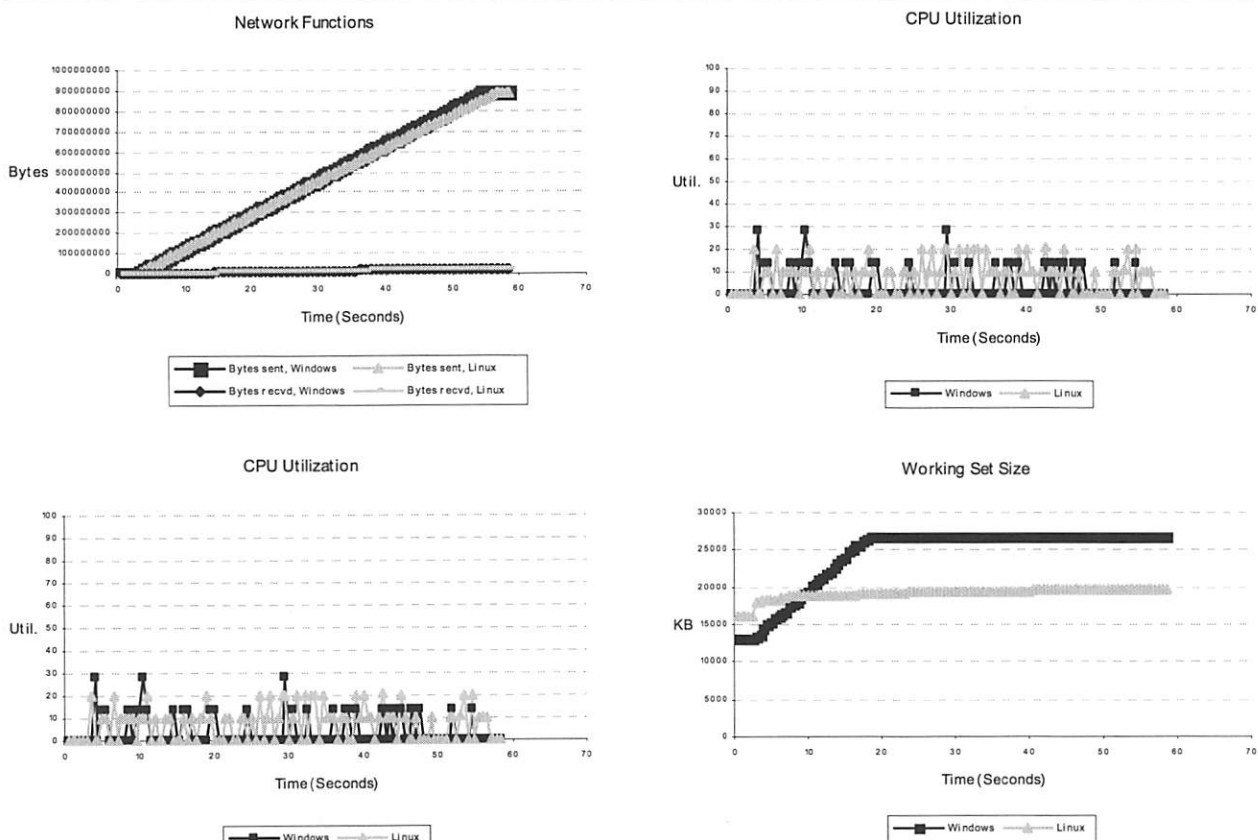


Figure 10: Results of Apache evaluation.

The network functions graph shows that each server was given a comparable load. Each server both sent and received the same amount of data. The CPU utilization graph was not unusual; the IO-bound Apache process did not heavily tax the CPU on either platform.

The memory graphs, however, were more telling. The Linux build of Apache with the worker MPM was able to run in a much smaller footprint, and, more importantly, was able to respond to the increased load quickly and without a large increase in either its working set or frequency of page faults. The Linux server's working set rose a small amount at the instant the load test began and remained steady throughout, while the Windows server took nearly twenty seconds to reach a steady state. During those twenty seconds, the Windows server was constantly causing page faults. This performance discrepancy could have been caused by a combination of configuration differences and implementation variations in the platform-dependent Windows and Linux MPMs.

Related Work

PAPI [3] is a system that shares several design goals with CAMP, but it accomplishes a slightly different task: it provides a platform-independent interface into *hardware* performance counters rather than operating system counters. On most desktop architectures, the underlying architecture is capable of providing controllable metrics about processor and low-level cache events. Each architecture has a different set of assembly instructions for accessing these counters, and each set has different semantics.

PAPI provides a platform independent interface to these counters through a standard C and FORTRAN interface, but it requires operating system kernel augmentation on most common platforms. PAPI does provide accurate metrics: its hardware counter interface has been used to validate other benchmarks [15].

Unlike CAMP, performance data provided by PAPI is difficult to correlate with a specific process. Utilities that allow this behavior either ignore the effects of scheduling or instrument the operating system kernels to report scheduling events [14].

Many system monitoring solutions are implemented using the SNMP protocol [23]. It provides a simple request/response protocol for system monitoring and management. SNMP is an application level protocol, implemented using UDP, that allows clients to host a set of named data. Examples of managed data include host name, uptime, and load. Because the protocol is at the application level, the local client implementation (the SNMP agent) is in full control of what data is reported and how it is obtained.

The latest version [17, 8] of the protocol provides bulk request/response functionality that scales more reliably with large distributed systems. When used with common object names, SNMP is capable of

providing a common interface into operating system performance data by means of a common network packet. However, the SNMP agent that runs locally on the monitored machine and services requests must still be able to actually *collect* the system performance data, which in most cases must be accomplished with native code. CAMP can provide a simple method for implementing a platform-independent SNMP agent that provides performance data for a set of managed hosts.

The Gloperf system [11] is part of the Globus GRID computing toolkit [6]. By using the GRID framework, it is able to measure selected performance statistics in a platform independent manner. The client side of this system runs as a daemon, and it actively collects its own statistics rather than broadcasting operating system or network protocol statistics. It uses a sensor/collection model in which the user installs a "sensor" on the client to be probed and periodically collects data. CAMP follows a completely different model: it only reports data that is already available on the host.

The Tau [20] set of tools is a high performance computing testing framework that has been recently updated to collect full statistics from a Java Virtual Machine [19], introducing platform independence. This has been used to create a Java profiler that can selectively instrument and measure parallel and distributed Java applications. The tool's feature set is comprehensive, and its standard interface allows it to be used on a number of platforms. However, all measurement is confined to within the virtual machine – Tau cannot measure system-level statistics.

CoMon [16] is a monitoring layer for the PlanetLab testbed [4]. It collects data from individual PlanetLab distributed nodes, and its concept of a "node-centric daemon" can deliver nearly all of the performance data that CAMP would provide. However, this tool only works with the PlanetLab operating system (a modified version of Fedora Core Linux), which makes the research less useful for systems outside of that controlled domain.

Future Work and Conclusions

CAMP's most important task is its continued implementation on other Python-supported platforms. Many UNIX-like operating systems are able to support both Python and the performance information necessary to support CAMP, including the newest Macintosh operating system.

There are endless possibilities for derived, second-layer functions. However, a standard set, including functions like rate calculators and aggregate network traffic, is feasible.

Distributed system performance testing is a broad problem, and CAMP seeks to solve the lowest level issue: CAMP allows developers to collect performance data from multiple platforms in a consistent, correct, and predictable manner.

Software Availability

CAMP can be obtained at: <http://wiki.csc.calpoly.edu/camp>.

Bibliography

- [1] Aguilera, M. K., J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 74-89, ACM Press, New York, NY, USA, 2003.
- [2] Annis, W., *pykstat: kstat API for Python*, 2001, <http://www.biostat.wisc.edu/annis/creations/pykstat.html>.
- [3] Browne, S., J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, p. 42, IEEE Computer Society, Washington, DC, USA, 2000.
- [4] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An Overlay Testbed for Broad-Coverage Services," *SIGCOMM Computer Communications Review*, Vol. 33, Num. 3, pp. 3-12, 2003.
- [5] Cusumano, M. A. and D. B. Yoffie, "What Netscape Learned from Cross-Platform Software Development," *Communications of the ACM*, Vol. 42, Num. 10, pp. 72-78, 1999.
- [6] Foster, I., C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal High Performance Computing Applications*, Vol. 15, Num. 3, pp. 200-222, 2001.
- [7] Gunter, D., B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic Monitoring of High-Performance Distributed Applications," *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*, p. 163, IEEE Computer Society, Washington, DC, USA, 2002.
- [8] Harrington, D., R. Presuhn, and B. Wijnen, *RFC 3411: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*, December, 2002.
- [9] Huang, Y., *pio: Solaris Process I/O*, 2004, <http://www.stormloader.com/yonghuang/freeware/pio.html>.
- [10] Knuth, D., *The Art of Computer Programming*, Vol. 2, p. 232, Addison-Wesley Professional, Third Edition, 1997.
- [11] Lee, C. A., J. Stepanek, R. Wolski, C. Kesselman, and I. Foster, "A Network Performance Tool for Grid Environments," *Supercomputing '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM)*, p. 4, ACM Press, New York, NY, USA, 1999.
- [12] Miles, C., "System Monitoring, Messaging, and Notification," *Proceedings of SAGE-AU'99*, The System Administrators Guild of Australia, Jul, 1999.
- [13] Mosberger, D. and T. Jin, "httpperf - A Tool for Measuring Web Server Performance," *SIGMETRICS Performance Evaluation Review*, Vol. 26, Num. 3, pp. 31-37, 1998.
- [14] Mucci, P. J., *papiex: Transparently Measure Hardware Performance Events of an Application with PAPI*, Innovative Computing Laboratory, University of Tennessee, 2005, <http://icl.cs.utk.edu/mucci/papiex/>.
- [15] Najafzadeh, H. and S. Chaiken, "Towards a Framework for Source Code Instrumentation Measurement Validation," *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pp. 123-130, ACM Press, New York, NY, USA, 2005.
- [16] Park, K. and V. S. Pai, "Comon: A Mostly-Scalable Monitoring System for Planetlab," *SIGOPS Operating Systems Review*, Vol. 40, Num. 1, pp. 65-74, 2006.
- [17] Presuhn, R., *RFC 3418: Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)*, December, 2002.
- [18] Renesse, R. V., K. P. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," *ACM Transactions Computing Systems*, Vol. 21, Num. 2, pp. 164-206, 2003.
- [19] Shende, S. and A. D. Malony, "Integration and Applications of the Tau Performance System in Parallel Java Environments," *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pp. 87-96, ACM Press, New York, NY, USA, 2001.
- [20] Shende, S. S. and A. D. Malony, "The Tau Parallel Performance System," *Int. Journal High Performance Computing Applications*, Vol. 20, Num. 2, pp. 287-311, 2006.
- [21] Sprunt, B., "Pentium 4 Performance-Monitoring Features," *IEEE Micro*, Vol. 22, Num. 4, pp. 72-82, Jul, 2002.
- [22] Sun Microsystems, *AWT: The Java Abstract Windowing Toolkit*, GUI development component of the Java 2, Standard Edition Platform, 1999, <http://java.sun.com/products/jdk/awt>.
- [23] Teegan, H., "Distributed Performance Monitoring Using SNMP V2," *IEEE Network Operations and Management Symposium*, Vol. 2, pp. 616-619, Apr, 1996.

Application Buffer-Cache Management for Performance: Running the World's Largest MRTG

David Plonka, Archit Gupta, and Dale Carder – University of Wisconsin-Madison

ABSTRACT

An operating system's readahead and buffer-cache behaviors can significantly impact application performance; most often these better performance, but occasionally they worsen it. To avoid unintended I/O latencies, many database systems sidestep these OS features by minimizing or eliminating application file I/O. However, network traffic measurement applications are commonly built instead atop a high-performance file-based database: the Round Robin Database (RRD) Tool. While RRD is successful, experience has led the network operations community to believe that its scalability is limited to tens of thousands of, or perhaps one hundred thousand, RRD files on a single system, keeping it from being used to measure the largest managed networks today. We identify the bottleneck responsible for that experience and present two approaches to overcome it.

In this paper, we provide a method and tools to expose the readahead and buffer-cache behaviors that are otherwise hidden from the user. We apply our method to a very large network traffic measurement system that experiences scalability problems and determine the performance bottleneck to be unnecessary disk reads, and page faults, due to the default readahead behavior. We develop both a simulation and an analytical model of the performance-limiting page fault rate for RRD file updates. We develop and evaluate two approaches that alleviate this problem: *application advice* to disable readahead and *application-level caching*. We demonstrate their effectiveness by configuring and operating the world's largest¹ Multi-Router Traffic Grapher (MRTG), with approximately 320,000 RRD files, and over half a million data points measured every five minutes. Conservatively, our techniques approximately triple the capacity of very large MRTG and other RRD-based measurement systems.

Introduction

Sometimes common case optimizations by the operating system can adversely affect an application's performance instead of improving it. For instance, in most OS, readahead intends to optimize sequential file access by reading file content into buffer-cache with the expectation that it will soon be referenced. In this paper, we identify and remedy a situation in which the performance of a popular time series database, the Round Robin Database (RRD) Tool, is adversely affected by the default OS readahead and caching behaviors.

We present an investigative method to discover the RRD system's performance bottleneck and an analysis of the bottleneck identified: the OS default file readahead and caching behavior. We describe two approaches to optimize system resource usage for maximum performance: (i) *application advice* to the OS to disable readahead and (ii) *application-level caching*. We validate our results by configuring and operating a Multi-Router Traffic Grapher (MRTG) system that performs over a half a million measurements

every five minutes and records them into a set of 320,000 RRD files in near real-time. We identify the additional factors that limit further scalability of the RRD system after these improvements. We also discuss OS improvements to the readahead behavior that could generally avoid the application performance problem we observed.

We investigate the scalability issues in a real world scenario. During the deployment of new network equipment (routers and switches) over the past few years at our university, the number of managed devices grew significantly, nearly doubling each year. This required our network measurement system's capacity to scale similarly. Today, for approximately 60,000 measured network interfaces, about 160,000 RRD files need to be updated every five minutes. These record interface byte, packet, and error rates. As the number of measurement points grew with the network size, we found that an increasing number of measurements did not get recorded into the database within the required five minute interval (20 to 80 percent failures).

We are motivated to study the scalability issues of RRD for two main reasons: (i) we were confounded

¹Based on the authors' experience in the MRTG and RRD-Tool user community

by our system's poor performance given that it is generously-sized with respect to processor, memory, and disk, and (ii) any performance gains achieved would benefit many, given the popularity of RRDTool. To the first point, our prior understanding of RRD file structure and access patterns led us to believe the amount of work should not overwhelm our system. RRD files are organized in such a way that a small number of blocks are accessed per update cycle. The set of blocks in a series of updates has a low entropy: that is, most updates touch the same set of blocks. Thus, we were of the opinion that the "working set" of blocks for the RRD files in our system could reside completely in the OS file buffer-cache. Unexpectedly, our system's CPU spent the majority of its time in an "I/O wait" state due to disk reads.

To study the state of the buffer-cache, we wrote a tool called *fincore* that exposes the cache "footprint" of a given set of files; it takes a snapshot of the set of file blocks or pages in the buffer cache. This helps us determine at any given time what pages were brought in memory by the OS and helps us discover the read-ahead effects. We are also able to study the average number of pages per file brought into the memory by an unmodified MRTG. This helps us determine the maximum number of RRD files the system could handle with fixed hardware resources. We wrote another tool called *fadvice* that can advise the operating system about the file access pattern using the *posix_fadvise* system call. This tool enables the user to forcibly evict any file's pages from the buffer-cache, providing a key function in controlled experiments.

Our work makes the following contributions:

- We provide two tools and a methodology to study buffering behavior. These enable a system administrator or analyst to study the buffer-cache of *any system* (provided it implements the requisite APIs) and draw conclusions about read-ahead behavior, cache eviction policies, and system capacity.
- We develop an analytical model and simulation that determine the number of RRD files that can be managed given fixed memory resources or to determine the memory required for managing a given number of RRD files.
- We present two optimizations to RRDTool and evaluate their performance and scalability. The first employs *application-level buffering or caching* to coalesce file updates. The second offers *application advice* to the operating system that RRD files are accessed randomly rather than sequentially, thus causing read-ahead to be disabled.

The remainder of this paper is organized thusly: We first provide background on MRTG and RRD, and introduce our network measurement system. Next, our investigation technique is described in the "Method and Tools" section. Then two complementary performance optimizations to RRDTool are presented in

the and "Application-Offered Advice" sections. The subsequent "Analysis" section contains our analytical model and simulation details. Ultimately, in the "Scalability" section, we report the scalability of the optimization techniques by running what we suggest is the world's largest MRTG on a single server. Therein we also discuss the factors limiting the further scalability of RRDTool after these improvements. The "Related Work" and "Discussion and Future Work" sections follow and we close with our conclusions.

Overview of MRTG and RRD

The Multi-Router Traffic Grapher (MRTG) is a perl script that collects network measurements and stores them in RRD files. Figure 1 shows a simplified MRTG in pseudo-code form. MRTG performance is satisfactory as long as it can consistently complete one loop iteration, consisting of one "poll targets" and one "write targets" phase, in less than the update interval, typically five minutes (300 seconds.)

```
# read configuration file
# to learn targets
readConfiguration();
do {
    # POLL TARGETS:
    # collect values via SNMP:
    readTargets();
    # WRITE TARGETS:
    # update values in RRD files:
    foreach my $target (@targets) {
        RRDs::update(...);
    }
    sleep (...); # sleep balance
                # of 300 seconds
} while (1); # forever
```

Figure 1: The MRTG daemon in pseudo-code.

MRTG refers to the configurable metrics it collects as *Targets*. Each target consists of two objects collected via SNMP, typically one inbound and one outbound measurement for a given network interface, i.e., a router or switch port. Thus, the number of targets per network device is typically a function of its number of interfaces.

The paired objects are each referred to as a *Data Source* (DS) in a *Round Robin Database* (RRD). The RRD file name itself and the file's Data Sources define the database "columns." *Round Robin Archives* (RRAs), or tables of values observed at points in time, are the database "rows." Figure 2 shows a typical RRD file managed by MRTG.

RRD performance is influenced by the RRAs defined within a file. Each RRA has an associated consolidation function, such as AVERAGE or MAX, that operates on a set of one or more *Primary Data Points* (PDPs), i.e., data points collected at the measurement interval. Additional RRAs typically require additional

work to be done periodically, such as on every half hour, two hours and one day. These aggregation times are defined as offsets from zero hours UTC. Thus all like-configured MRTG RRD files require aggregations to be done every half hour, more every two hours, and then the most aggregations at midnight.

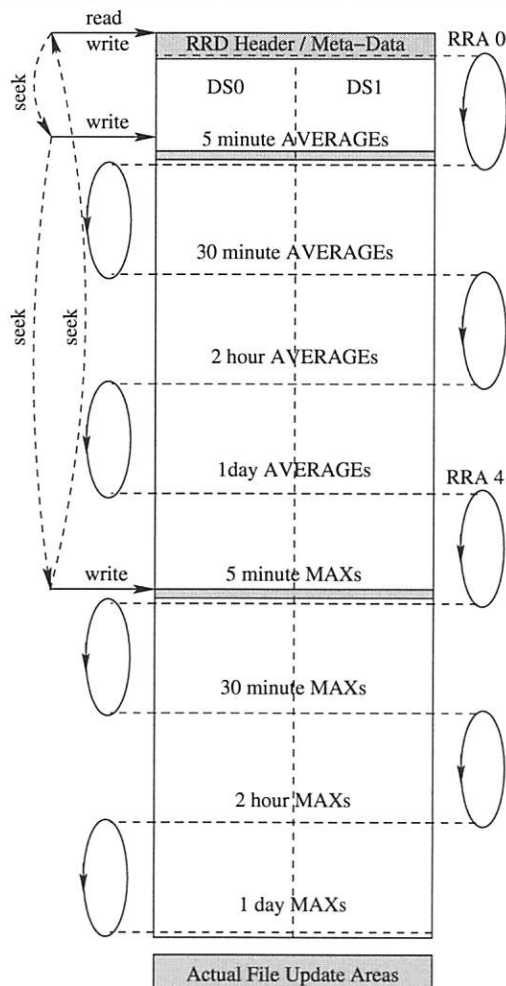


Figure 2: A typical MRTG RRD file and update operation. This RRD file stores two Data Sources (DSes) in eight Round Robin Archives (RRAs): four AVERAGE and four MAX RRAs. The 5 minute AVERAGE and 5 minute MAX RRAs are being updated.

Our MRTG System

Our MRTG system use RRDTool and currently measures approximately 3,000 network devices. Primarily, the devices are switches and routers in our campus network including those in the core and distribution layers and most of the network equipment at the access layer, serving users in approximately 200 campus buildings.

In this work, we refer to this production MRTG network measurement system as the System Under Test (SUT.) The SUT's characteristics including its

software are summarized in Table 1.² The system's page size is 4KB and our file-systems are configured with a 4KB block size. Thus, we will conveniently use the terms "block" and "page" interchangeably when referring to segments of a file whether they are on disk or in memory.

Component	Characteristics
Processors	8 × Intel Xeon @ 2.7 GHz
Processor Cache	2 MB
Memory	16 GB
Disk	SAN: RAID-10, 16 × 2 disks
Operating System	Linux 2.6.9
File System	ext3 and ext2, 4KB blocksize
I/O Scheduler	Deadline
Software	Version
MRTG	mrtg-2.10.5
RRDTool	rrdtool-1.0.49

Table 1: Characteristics of the System Under Test and its software.

As our centrally-manged network has grown, our MRTG system has grown in terms of computing power and storage. One significant technique we employ to improve MRTG's scalability is to divide the targets amongst a configurable number of MRTG daemons that we increase as our number of targets increases; we process about 10,000 targets per daemon. So, our one MRTG "instance" is actually a collection of MRTG daemons running on one server. Another dimension in which our MRTG system is larger than most is that we resize RRA 0 (the five minute averages) to store up to one year or five years of data. This increases an MRTG RRD file's size from the typical 103 KB to 1.7 MB or 8.2 MB, respectively, of course requiring much more disk space. (We see that this does not adversely affect performance in the "Analysis" section.)

Prior to this work, our network growth exceeded the scalability of the SUT. The Appendix lists system and MRTG configuration recommendations that we've tested and used in our system to meet our performance goals.

Method and Tools

Examining System Activity

We started by examining the SUT's activity to determine the nature and extent of the performance problem. We present three measurements that led us to the root cause of the problem and that allow us to evaluate potential solutions.

First, we measured the time to completion of each measurement interval by each MRTG daemon on

²MRTG 2.10.5 patched as follows: Modified fork code to use select as in mrtg-2.10.6. Added a --debug=time option to report poll targets and write targets times. Removed test for legacy ".log" files (log2rrd), and threshcheck.

our system, with approximately 160,000 targets in total. As shown in Figure 1, this consists of two phases, first polling the network statistics via SNMP and then updating the pertinent RRD files. Figure 3 is a scatter plot with the measurement's time of day on

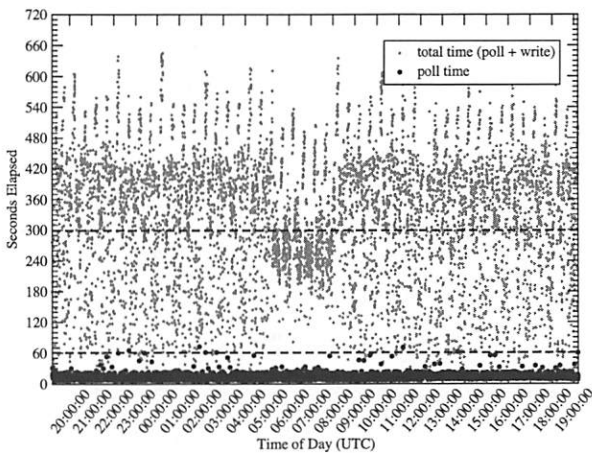


Figure 3: Original MRTG performance on the SUT with 160,000 targets processed by 28 MRTG daemons. The total time for poll and write targets phases often exceeds the five minute performance goal.

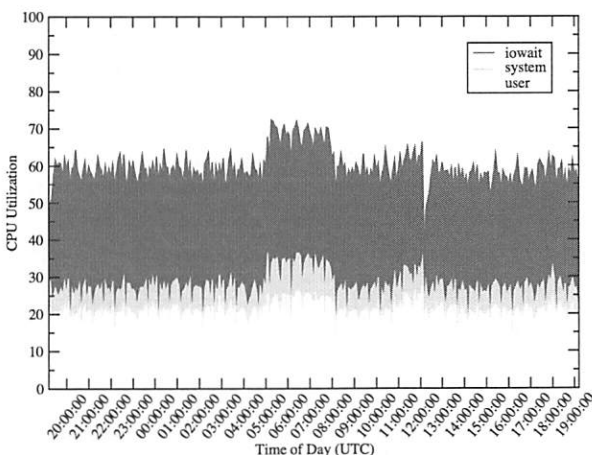


Figure 4: Original CPU utilization on the SUT with 160,000 targets processed by 28 MRTG daemons. CPU I/O wait state is excessive due to page faults for RRD file content.

the horizontal axis and the seconds elapsed on the vertical axis. The “poll time” dots show the time taken (in seconds) to poll the network and the “total time” dots signify the time taken by the poll and the subsequent write phase in one loop iteration by an MRTG daemon. Note that all the network polling finishes well below 60 seconds (marked by a horizontal line.) The writing phases very often do not finish within the period of 300 seconds (our five minute performance goal, also marked with a horizontal line) for the daemons; some even take 10 minutes to complete. This is

clearly unacceptable performance because it delays the measurements for the subsequent poll phase in the single-threaded MRTG daemon, resulting in missing measurements.

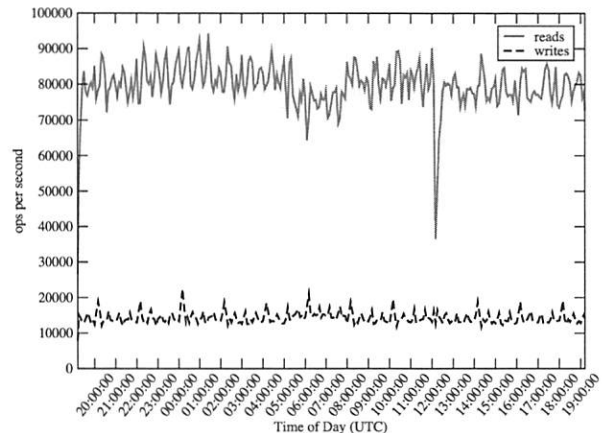


Figure 5: Original Disk utilization on the SUT with 160,000 targets processed by 28 MRTG daemons. Block read operations unexpectedly exceed write operations on RRD file updates.

Further examination reveals that the CPU was in I/O wait state for a majority of the time. Figure 4 shows the CPU utilization. The user and system CPU utilization levels are ~20% and ~10%, respectively, and are not the bottleneck. However, the CPU is spending more than half its time in the I/O wait state. Thus, the CPU wastes most of the time waiting for I/O to complete.

To understand the I/O wait, we studied the actual number of reads and writes involving the disk. Unexpectedly, the system was doing close to 90,000 reads per second (see Figure 5.) In contrast, the number of writes was stable at ~12,000 writes per second. The high number of reads suggests that files are not being cached effectively. This led us to examine the contents of the buffer-cache to determine why.

Examining Buffer-Cache Content

The system's buffer-cache content gives a good indication of which files' accesses are benefiting from caching in core memory. Unfortunately, a system's buffer-cache content is generally hidden from users and user processes. Prior work has resorted to timing file block accesses to surmise whether or not a given page already resides in the buffer-cache memory [4]. While suitable in some situations, this technique is indirect and has the unwanted side-effect of modifying the cache because it references the pages about which it inquires, causing them to be brought into cache and likely evicting other pages. Thus, a user tool to passively investigate buffer-cache content is desired.

We introduce a new user command called `fincore` that is used to determine which segments of a file are in core memory, presumably because they reside in the

buffer-cache. The `fincore` command takes file names as arguments and displays information about file blocks or pages in memory. The `fincore` command uses two common system calls to accomplish this: `mmap` and `mincore`. That is, it first maps a file into its process' address space, then asks which pages of that segment of the process' address space are in core at that time.³ Using `fincore` we were able to uncover the readahead effects on the buffer-cache. As an optimization, Linux reads pages ahead from the disk into the buffer, anticipating locality of subsequent reads. This improves performance for most applications by decreasing subsequent read latencies. With the current implementation of RRDTool, the readahead can have a highly adverse impact on performance and scalability. A brief discussion of the readahead algorithm within the context of the RRD file shown in Figure 2 can make this clearer.

The readahead algorithm tries to guess whether the file being accessed is going to be read sequentially (when readahead is actually useful) or randomly. On an RRD file update, the first read is for the meta-data at file offset zero, i.e., the beginning of the file. The maximum readahead window size is 32 blocks for `ext2` and `ext3`, and, on an initial read, the readahead window starts at half that maximum in anticipation of sequential access. So, 16 pages are read into buffer-cache when the application read just one. The file offset of the second block needed to update the AVERAGE RRA depends on the current update position within the RRA.

For the typical MRTG RRD file, this will lie within the first 16 pages. The file offset of the third block needed for the MAX RRA update sometimes lies beyond the first 16 pages which can lead to further 8 pages being read in to the memory. (Eight pages are read as the readahead algorithm reduces the readahead window at the random seek into the MAX RRA.) These file block accesses are depicted in Figure 6.

A typical RRD file update consists of two RRA updates (AVERAGE and MAX). With the default readahead, most blocks that are read in are unnecessary. In the event of data kept over a longer period of time, as in our case with five minute averages for one year or five years, the write for the AVERAGE RRA often lays well beyond the first 16 pages. The readahead window is reduced to 8 pages for the next random read for the AVERAGE RRA update and then to 4 for the subsequent random read for the MAX RRA update. The readahead algorithm starts to adapt to the random reads by reducing the readahead window. A typical RRD file update requires just three block updates, yet we end up bringing 28 (16+8+4) blocks into the file cache. The file is then closed which causes the adapted readahead value to be lost, reverting to 16 the next time the file is opened. For the

³`fincore` is not entirely passive; it likely affects the cache slightly because it opens the file and thus causes an access to its inode block.

typical RRD file with 800 recorded values, we end up bringing almost the full file into cache. If we could bring just the required "hot" blocks into the file cache by suppressing readahead from the beginning, we would get better performance and scalability.

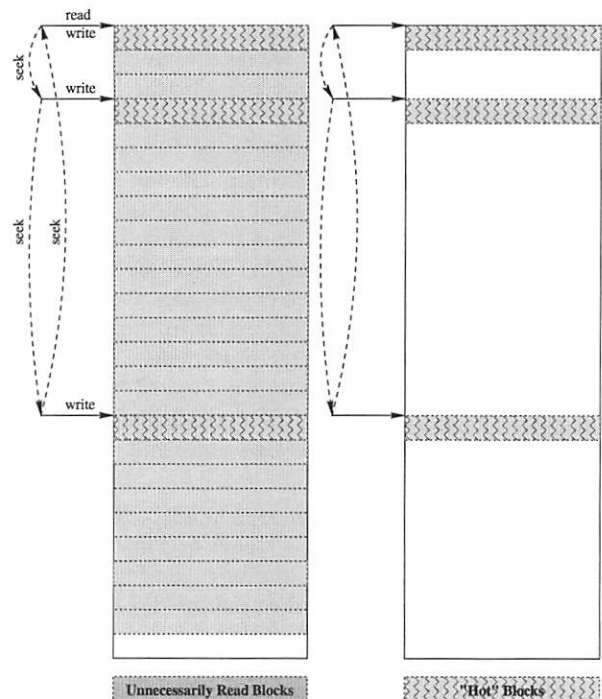


Figure 6: A sample RRD file update operation and the blocks involved. Readahead causes many blocks to be read unnecessarily, rather than just the "hot" blocks.

We see that the number of blocks most often required per MRTG RRD file per update is three. Based on our `fincore` observations (in the "Analysis" section), we note that there is also a low "churn" rate of these blocks. That is, once fetched into memory, these blocks were useful for a long period of time. If we were caching only the required blocks, there would be no waiting on reads, eliminating our performance problem.

Due to the default readahead behavior, we must wait for reads from the disk since we find almost nothing useful in cache. This is because, for instance for the file cache to accommodate 300,000 RRD files, $300,000 \times 24 \times 4$ KB (close to 30 GB of memory for the cache) are required. Since the file cache isn't that large, the page replacement policy evicts the pages that will be needed later. However, with suppressed readahead, we would cache just three blocks per file: $300,000 \times 3 \times 4$ KB (3.6 GB) and an order of magnitude less memory is required to fit everything desired in file cache. Actual buffer-cache behavior for RRD files is not quite as simple as this example; see the "Analysis" section for details.

Evicting Buffer-Cache Content

For repeatable experiments involving the buffer-cache, we require fine-grained control over the buffer-cache content. For instance, one run of an experiment such as an RRDTool update, brings pages of the RRD file from the disk into the buffer-cache. If we wish to see the effects of a subsequent update (reading the pages again from the disk), we need to evict the pages that were brought in earlier. Generally, the only methods available to forcibly evict pages from the buffer-cache were to either (i) unmount the file-system containing the cached files or (ii) populate the cache with hotter pages by accessing other content more frequently or more recently, thus invoking the systems page replacement algorithm to evict the unwanted pages. To perform controlled experiments we wanted a more convenient method for a user to forcibly evict specific files' blocks from the buffer-cache. To do so, we introduce a new user command called `fadvice` that is used to provide file advisory information to the operating system. The `fadvice` command takes file names as arguments.

Our typical use of `fadvice` is to advise the system that we "don't need" a file's blocks and that we'd like them to be evicted from the buffer-cache.⁴ In this case, the file is first synchronized so that its dirty pages are transferred to the storage device, e.g., the disk, and then the advice is issued. The `fadvice` command uses the `fsync` then `fadvice` system calls to accomplish this.

Application-Level Buffering

We've described our system and showed that it does not meet our performance goal. A number of

⁴The Linux 2.6.9 source code and experimentation show that `fadvice DONTNEED` immediately evicts non-dirty pages from the buffer-cache. Other implementations might not evict the pages immediately.

RRDTool users have proposed significant modifications to RRD measurement systems to improve performance by modifying I/O behavior [22, 12, 9]. These proposals generally involve intercepting RRD file updates and recording them to be written later. The updates are thus deferred, then later coalesced and written. The result is improved performance by the introduction of an independent thread to perform application writes and by the better locality characteristics of these periodic, coalesced writes.

Since this essentially implements a buffer-cache within the application, we call this technique *application-level buffering*.

Technique

We now describe our application-level buffering implementation called `RRDCache`, shown in Figure 7. `RRDCache` has three main components:

1. The `RRDCache Library`: a perl module that handles an application's calls to RRDTool's library. Specifically, it is used in place of RRDs perl module and provides the same functions, i.e., `update`, `graph`, etc.
2. The `RRDCache Journal Buffer`: a tmpfs file-system [24] to which updates are temporarily stored sequentially. (This is reminiscent of a journal in a journaling file-system.) We selected a memory-based file-system because it reserves a portion of memory exclusively for RRD and completely eliminates disk I/O during the RRD update operation.⁵
3. The `RRDCachewriter`: a script scheduled hourly using cron that periodically organizes updates and applies them to the RRD files on disk. In

⁵The `RRDCache` journal buffer need not be a memory-based file-system; it could be a disk-based file-system and still yield improved performance due to the better locality characteristics of appended writes to files.

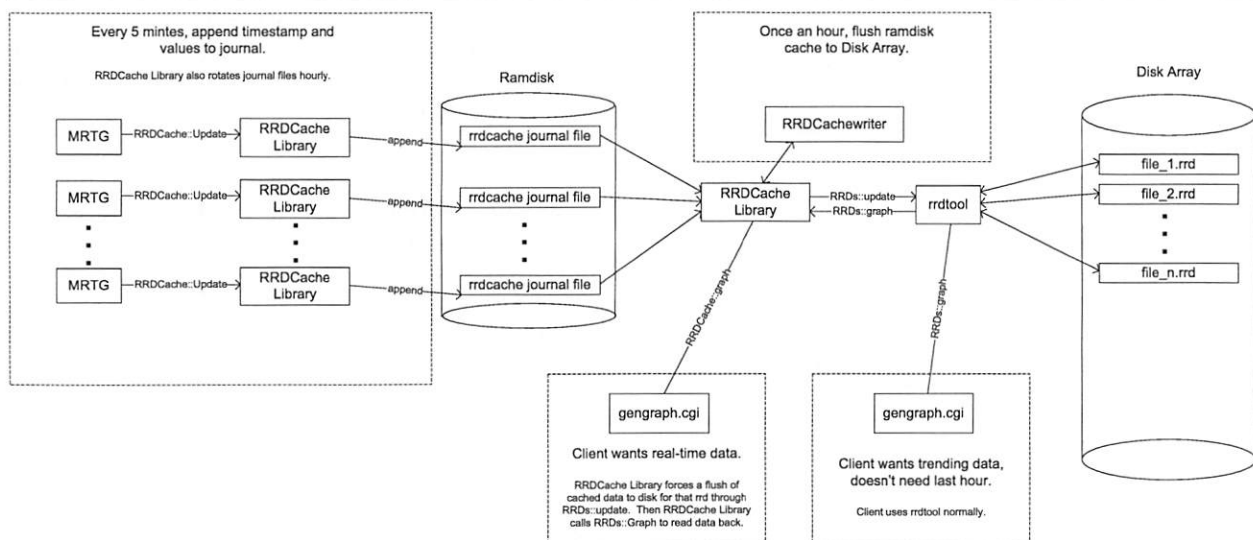


Figure 7: An Overview of `RRDCache`.

this way, RRDCachewriter performs both asynchronous writes and a sort of I/O scheduling on behalf of RRDCache and the application using it. This side-stepping of the operating system's default behavior, such as flushing dirty buffers to disk every five seconds, results in performance gains.

Normally an MRTG daemon (or any application that uses RRD files) accesses RRD files on disk directly by using the RRDTool API. With RRDCache, MRTG and other applications instead call functions in the RRDCache library. RRDCache presents the same API as the RRDTool. So, when an MRTG daemon calls the RRDCache:update function, the arguments are appended to a RRDCache journal file associated with calling process, i.e., the MRTG daemon.

The RRDCache journal file is located on the tmpfs file-system, eliminating disk I/O for the update. Periodically (once every hour), the RRDCachewriter runs to process any new data that has been written to the RRDCache journal files. The RRDCachewriter handles the updates from the journal files by committing the update to the appropriate RRD files on disk. In the process, it coalesces all the updates meant for a particular RRD file. If the requisite file pages are not already present in the buffer-cache, this has the benefit of bringing them into memory much less frequently. The RRDCachewriter can be run more often if the tmpfs runs out of space between runs. (We have been using 1 GB of our main memory for the tmpfs file-system and that has proven to be sufficient for the 160,000 targets polled at five minute intervals.)

Performance Impact

The performance of the measurement system using RRDCache is much improved. For our MRTG system, Figure 8 shows the results. We see that the MRTG daemons finish in well under 60 seconds which includes *both* the polling and writing. Contrast this with Figure 3, in which most of the updates were not achieving even the five minute performance goal.

Since the RRD file updates were performed by RRDCachewriter once an hour, and ordered by RRD file, there is a limited amount of I/O wait by the CPU at the start of every hour (Figure 8). This I/O wait is much less than the original system's I/O wait shown in Figure 4. The CPU utilization by the user and system processes remains the same as before.

Also evident in Figure 8 are spikes in disk read and write activity once per hour as the updates are being transferred from the journal buffer to the RRD files on disk. These disk I/O rates are much lower than the original system's rates shown in Figure 5.

One complication of RRDCache's technique is that the application-level journal buffer is not readable by RRD applications other than the RRDCachewriter; currently it is just a buffer for writing, not a cache for reading. While updates reside in the RRDCache journal buffer, they can't be directly accessed by applications

that may wish to graph recent measurements, for instance. Thus RRDCache slightly changes near real-time access semantics for RRD files. To work around this, RRDCache provides a graph function that immediately flushes pending updates from the journal buffer into RRD files upon attempts to read them and then returns the result of the RRDs::graph function. Applications then have the option of accessing RRD files directly through the RRDs interface, thus reading perhaps only older data suitable for trend analysis. However, performance would degrade if applications were to read every RRD file once per update interval (e.g., five minutes) to retrieve the most recent measurements, reverting to approximately the poor performance originally observed. If ever this becomes a problem, RRDCache could be improved so that its journal buffer is a true buffer-cache, consistent amongst both reading and writing processes.⁶

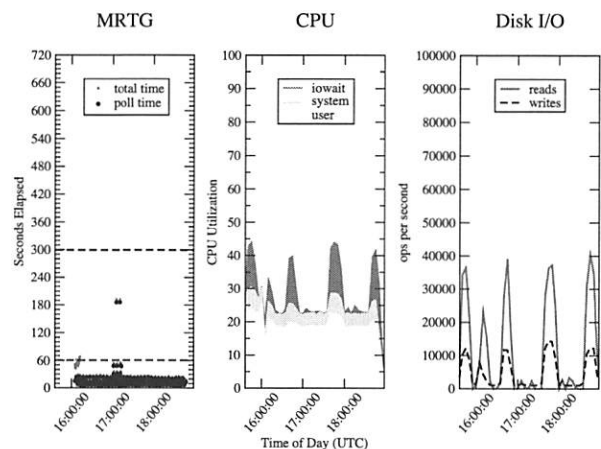


Figure 8: RRDCache: Performance on the SUT with 161,000 targets processed by 27 MRTG daemons. The five minute performance goal is easily met although CPU I/O wait and excessive block reads vs. writes are evident.

Application-Offered Advice

We find sufficient motivation to avoid the operating system default readahead and buffer-cache behaviors because of the latencies observed while updating RRD files. In the previous section, we've shown that modifications that drastically reduce RRDTool's file I/O can achieve better performance by *working around* the operating system's default behavior.

In this section, we instead improve performance by *directly influencing* the operating system behavior. Specifically, we identify a mechanism to cause just the desired RRD file blocks to be read and cached.

Technique

One technique to suppress readahead is to use the `posix_fadvise` system call. This allows applications

⁶Maintaining the journal buffer as a collection of very small RRD files would enable it to be read conveniently.

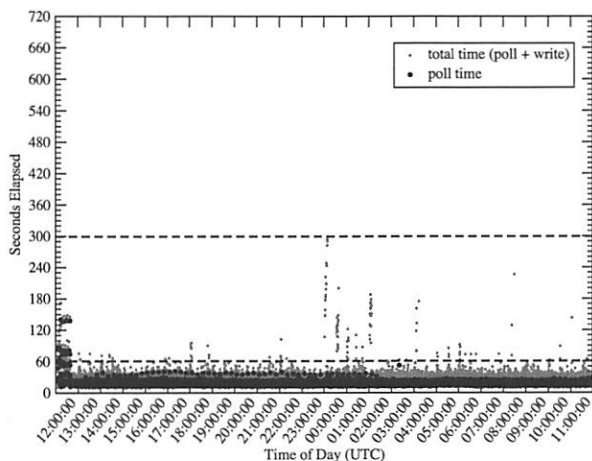


Figure 9: RRD with fadvise: MRTG performance on the SUT with 162,000 targets processed by 19 MRTG daemons. The five minute performance goal is clearly met.

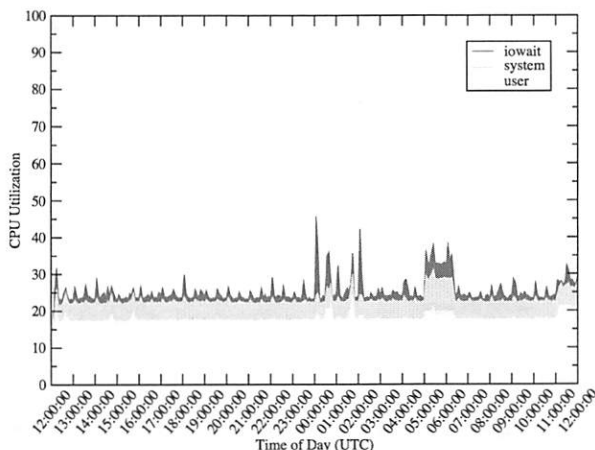


Figure 10: RRD with fadvise: CPU utilization on the SUT with 162,000 targets processed by 19 MRTG daemons. CPU I/O waits are minimal.

to advise the operating system of their future access patterns of files. The application identifies regions of an open file (by offset and length) and offers hints as to whether it will access them sequentially (the default) or randomly. Additionally, the application can inform the OS whether or not it expects to access those file regions again in the near future. Thus, for RRD files, we are able to advise the OS that the file accesses will be “random.” This turns off readahead. The result is that only the “hot” blocks shown in Figure 6 are read and cached. For Linux 2.6.9 with fadvise RANDOM, on a typical five minute update only three blocks are cached.

Performance Impact

The benefit of disabling readahead is realized immediately. Figure 9 shows the time elapsed per loop iteration of each MRTG daemon. With a system of 162,000 RRD files serviced by 19 MRTG daemons, we see that they finish within 60 seconds including

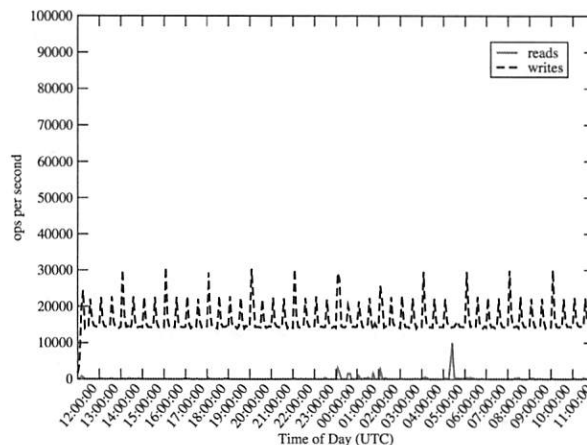


Figure 11: RRD with fadvise: Disk utilization on the SUT with 162,000 targets processed by 19 MRTG daemons. I/O Read operations are reduced to nearly zero due to efficient use of the buffer-cache.

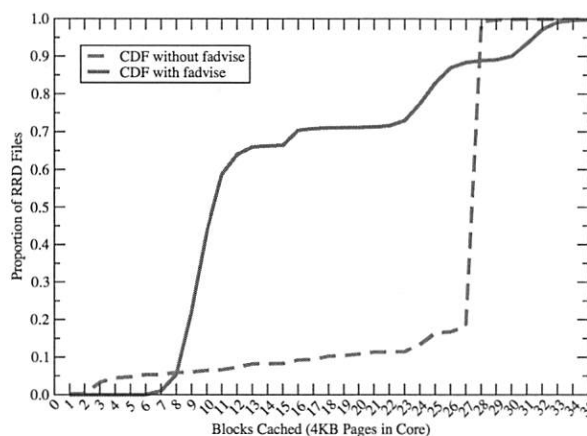


Figure 12: fincore: Comparison of proportion of RRD files by number of blocks cached without file access advice (before) and with advice (after). With fadvise RANDOM, unnecessary RRD file blocks no longer occupy the buffer-cache.

both the polling and writing phases. This is contrasted with Figure 3 where we see that most of the updates do not meet our five minute performance goal. This also suggests that we can monitor even more targets within a five minute interval. Note that the increase in the time to update at zero hours UTC is due to aggregation (of the AVERAGE and MAX values) that is synchronized across all RRD files, and a potential performance limitation that we also discuss. The performance potential and limitations are further explored in the “Scalability” section.

As expected, the CPU is largely freed from waiting on I/O to complete (Figure 10). Again comparing with Figure 4, we note that the CPU utilization by the user level processes and the system remains the same as before. The significant difference is the low amount of waiting on I/O by the system when the readahead is

suppressed. This implies a lowered number of reads with the buffer-cache becoming much more effective in caching the needed blocks. This is validated by our measurement of the reads issued to the disk per second by the system (Figure 11). The performance gain is significant, reducing approximately 90,000 reads per second to about 100 reads per second.

Figure 12 shows the plot of the proportion of RRD files vs. the number of blocks cached for both the original system without fadvise and the modified one with fadvise. One can see the sharp decrease in the number of blocks cached by the modified system with fadvise compared to the original system. For the original system, a sharp inflection point occurs at 27 pages. This indicates for a majority of the RRD files 27 pages were required. Also, the original system required more pages per file but couldn't fit them in the buffer-cache. For the modified system with fadvise, the inflection point occurs at 8, where the system requires 8 pages for most of the RRD files. Since the buffer-cache has space available for more pages, some of the RRD files get to keep more than 8 pages in the buffer-cache.

Analysis

To better understand the buffer-cache behavior when updating a very large number of RRD files in near real-time, we developed both an analytical model and a simulation. Analytical modeling improves our understanding of RRDTool's file update behavior so we have a solid foundation on which to propose general solutions. The resulting model also provides a convenient way to calculate expected page fault rates without experimentation and measurement. The simulation allows us to gather a broader set of results than either the model or real-world experiments that would prohibitively require repeated reconfiguration of a real system's RRD files or physical memory.

Analytical Model

We present an analytical model to predict the page fault rate given an RRD file configuration and an estimated number of pages available for the each RRD file in the system's buffer-cache memory. The modeling is done for a system with RRDTool patched to do fadvise RANDOM.

For this analytical model, we first need a list of the unique Primary Data Point (PDP) counts in increasing order, over which the consolidation is performed for every RRA (for AVERAGE, MAX, and so on). Recall that each RRA consolidates some number of PDPs that were gathered at the measurement interval, so an RRA's PDP count determines how often it is updated. For instance, for the RRD file shown in Figure 2, the ordered list of values of PDPs over all RRAs is: {1, 6, 24, 288}. These represent the periods of consolidation which in the case of Figure 2 refers to 5 minutes (1), 30 minutes (30/5 = 6), 2 hours (120/5 = 24) and 1 day (288). We also need a corresponding list

of number of RRAs that are configured to consolidate each of those numbers of PDPs. That is for 2, since both MAX and AVERAGE RRAs are kept for each of the PDP value, the associated count list for {1, 6, 24, 288} is {2, 2, 2, 2}.

We denote the ordered PDP list as $\{x_1, x_2, \dots, x_n\}$ and the associated RRA count list as $\{c_1, c_2, \dots, c_n\}$. The cardinality of the ordered list is n . Hence in the example, $\{x_1, x_2, x_3, x_4\} \equiv \{1, 6, 24, 288\}$ and $\{c_1, c_2, c_3, c_4\} \equiv \{2, 2, 2, 2\}$. Here, $n = 4$.

Now, for one update of an RRA, let B be the number of bytes written into the RRD file. In our case $B = 16$ bytes, for the two 8-byte floating-point values. For simplicity, we assume that each RRA is block aligned. This does not sacrifice generality since the average number of page faults due to crossing page boundaries is still predicted accurately as long as an RRA is at least one block in size, which is typically the case. The block size is S bytes. $S = 4096$ bytes in our case.

The number of updates that fit in a block is $u = S/B$. That is, after every S/B updates a page fault will occur. $u = 256$ in our case.

Let T be the time after which the primary data point is updated. $T = 5$ minutes in our case.

Now we estimate, for a single RRD file, the rate at which page faults occur given maximum p pages are available in the buffer-cache for use for this file (excluding the inode and indirect blocks.) When more than p pages are needed, LRU is used to evict pages to make place for newer ones.

The time estimated to a page fault, t , is the following:

$$t = \text{minimum}(uT, x_s T) \quad (1)$$

where $\exists s$ such that

$$p - 1 \geq \sum_{i=1}^s c_i \quad (2)$$

$$p - 1 < \sum_{i=1}^{s+1} c_i \quad (3)$$

$\text{minimum}()$ returns the lower of the two values.

The rationale behind $x_s T$: For each RRA, a page is needed in memory. So we calculate the count of RRAs which can be fit in $p - 1$ pages. (Only $p - 1$ pages are available to the RRAs because one page is required for the first block of the RRD file that is always read as it contains the RRD metadata.) The subscript s is used to index into the ordered list to determine the interval after which the fault will occur. Each index into the ordered list is a discrete point in time when a fault will occur. For instance, if $s = 2$, then x_2 implies fault would occur every half hour. (2) and (3) give the index s based on the count of RRAs that can fit in $p - 1$ pages.

The rationale behind uT : A page fault will surely occur after u updates.

The number of pages that will see a fault, m , after time t :

$$m = \delta \sum_{i=1}^n \frac{1}{x_i} \quad (4)$$

where:

$$\delta = p - \sum_{i=1}^s c_i \quad (5)$$

Rationale behind δ : The number of pages that need to be brought into memory is the count of the RRAs at a particular index s , which cannot be held by p pages.

Rationale behind $\sum_{i=1}^n \frac{1}{x_i}$: In our case, the update rate for x_1 is six times the update rate of x_2 . Therefore, the number of faults for $x_1 = \frac{1}{6} x_2$. We sum for all the fault rates through x_n relative to x_1 and hence the $\frac{1}{x_i}$ factor.

Rate of page fault, r , is given by m/t :

$$r = \frac{m}{t} \quad (6)$$

$$r = \frac{\delta \sum_{i=1}^n \frac{1}{x_i}}{\text{minimum}(uT, x_s T)} \quad (7)$$

This model essentially predicts the average values shown with no readahead in Figure 13, as verified by simulation. Thus, the model provides a quick way to calculate either the expected page fault rate given a buffer-cache memory constraint, or vice-versa. In addition to this practical result, the analytical model led us to the following insights:

- The total RRD file size is practically irrelevant. This is ideal since it frees us to extend our data retention arbitrarily, bounded only by disk space. For instance, recall that in our system we regularly resize our RRD files to store five minute averages for up to five years. (MRTG typically stores them for less than three days.) The resulting $17\times$ increase in file size only nominally affects the page fault rates.
- The total number of RRAs with a given aggregation level is important. For instance, removing the five minute MAX RRA,⁷ which duplicates the values in the five minute AVERAGE RRA, results in a significantly lower page fault rate when buffer-cache memory is scarce.

Simulation

In addition to deriving the analytical model, we developed a page fault simulation. This simulation provides a means by which to validate the average page fault rate predicted by the analytical model. It

⁷We suppose that the MRTG five minute MAX RRA exists for historical and convenience reasons. While it allows one to graph just the MAX values and see the entire time range, users typically put both the AVERAGE and MAX values on the same graph.

also exposes the distribution, variance, and peak page fault rates by time of day.

First we simulated an entire lifetime of an RRD file's updates using RRDTool itself, but with synthetic input data. Before each update, we used our `fadvise` command's "don't need" technique to evict the file's cached (hot) pages. After each update we used our `fincore` command's technique to determine the hot pages and recorded the page numbers to a log.

Secondly, we wrote a buffer-cache simulator with a Least-Recently-Used (LRU) page replacement policy and replayed the page operation log recorded earlier to determine the page faults with varying numbers of buffer-cache pages being available per RRD file. While our SUT's buffer-cache is actually managed using the Linux 2.6 page replacement algorithm [8], similar to 2Q [10], we make the simplifying assumption that LRU is suitably similar for the purpose of simulation. In addition to the RRD file data blocks, we also simulated access to the file's inode and indirect blocks. On ext2 and ext3 file systems, a typical MRTG RRD file incurs an indirect lookup (and therefore an indirect block must occupy space in the buffer-cache) for each data block above the twelfth since only the first twelve blocks are directly referenced in the inode.

From the resulting simulated behavior, we can determine the expected page faults for a single RRD file over time. We then extrapolate by multiplying by the target number of RRD files to determine what amount of buffer-cache (as limited by physical memory) reduces page fault disk reads to an acceptable level.

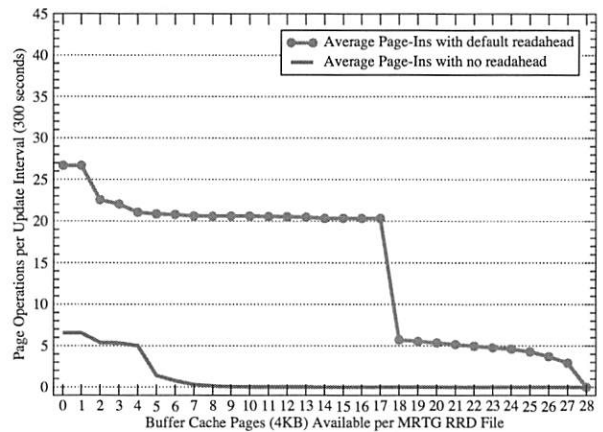


Figure 13: Simulation with and without `fadvise`: Page-Ins as a function of the number of buffer-cache pages available per MRTG RRD file. Significantly reduced paging rates result following the dramatic drops.

We run the simulation for both the original RRDTool (default readahead), and the one patched with `fadvise` RANDOM (no readahead). Figure 13 shows the average number of page-in operations for both versions of RRDTool as a function of the number of buffer-cache pages available per MRTG RRD file. For

the original RRDTool, when the number of pages available in the buffer-cache is less than 18, the number of page faults is very high. It falls at 18 because the 16 pages required for initial readahead are available in the buffer-cache at this point. (It is 16+2 because two extra blocks are required for the file inode and indirect blocks.)

Also, sometimes, the AVERAGE and MAX RRAs get written within these 16 pages. For the patched version with *fadvice* RANDOM, the average number of page-ins is close to zero if more than 7 pages are available in the buffer-cache. Page faults still occur when 8 pages are available in the buffer-cache but the average page-in rate is extremely low as shown in Figure 14. Note that more pages are written out at the aggregation intervals of 30 minutes, 2 hours and one day. The time of day 00:00 UTC shows the peak paging activity, when aggregation happens for daily RRAs.

Our simulation results are validated by the earlier observations of the real system's performance with *fadvice* RANDOM. Specifically, the read and write pattern of the simulation in Figure 14 agrees with the observations of the real system with *fadvice* RANDOM in Figure 11, i.e., near zero read (page-in fault) rate. Earlier, using *fincore* in the real system, we also observed that most RRD files had only 8 pages in cache (Figure 12); this is the value that simulation shows (Figure 13) is the minimum required to achieve an average page fault rate of nearly zero.

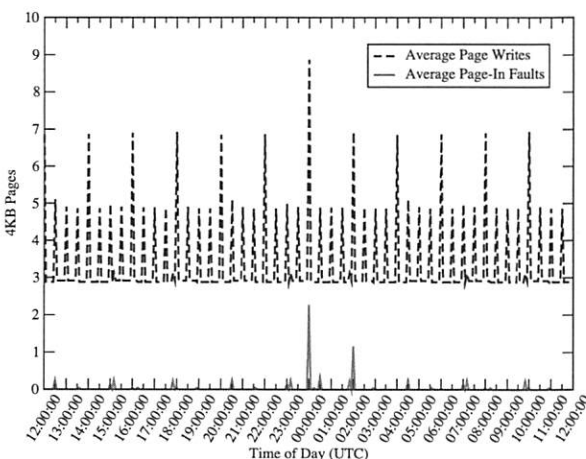


Figure 14: Simulation with *fadvice*: Average page-out and page-in operations by time of day given eight buffer-cache pages available per MRTG RRD file. As few as eight pages per file reduces page-ins to near zero.

Scalability

We have shown that using either *application-level buffering* or *application-offered advice* dramatically improves the performance of RRD systems. In this section we show the performance and capacity scalability characteristics of a very large RRD and

MRTG-based network measurement system by testing it first with just *application advice*, and secondly with *advice plus application-level buffering*. Thus, we explore the performance of the simpler of the two techniques and also the two techniques combined to determine the upper-bound to the scalability of such a system.

Our production MRTG system today monitors approximately 3,000 network devices with approximately 160,000 MRTG targets. Recall that each target is typically a pair of measurements such as byte, packet and error rates, inbound and outbound. Thus, the production system measures and records approximately 320,000 data points every five minutes. Having already found that either improvement technique results in satisfactory performance and thus does not push our system to its limit, we now construct an even larger system to study scalability.

We create an MRTG system three times the size by replicating our existing production measurement system so that there are three like-configured MRTG instances all running on one server. Our experimental procedure is to begin with approximately 160,000 (i.e., the production MRTG system) and then progressively add 20,000 targets every twenty minutes (10,000 from each of the replicated instances), until all three systems are running in parallel for a total exceeding 480,000 targets.

MRTG with *fadvice* RANDOM

We first tested the scalability of MRTG with RRDTool patched to do *fadvice* RANDOM. The performance results are shown in Figures 15 and 16. This is the system we claim as the world's largest MRTG, operating with acceptable performance at around 320,000 RRD files. While there are some outlier points in the upper left of Figure 15, they occur at twenty minute intervals and there are exactly two per interval. Thus, these outliers are an artifact of the experimental procedure showing latency during just the very first loop iteration of each of the two new MRTG daemons as their the set of hot pages for their RRD files are read into buffer-cache. Beyond about 320,000 targets in the scalability test, performance is unacceptable because page faults increased and CPU utilization continually exceeded 65%, leaving little room for other tasks.

In Figure 16 note the spikes in CPU I/O wait state just following 1600 and 1800 hours. These are due to aggregations that occur every two hours in typical MRTG RRD files. Furthermore, note that as the number of targets increases, similar spikes are seen at half hour intervals following 1800 hours. These spikes indicate that the number of hot pages exceeds the capacity of the buffer-cache on the SUT, resulting in an excessive page fault rate. We estimate our buffer-cache requirement to be 8 pages per file and $480,000 \times 8 \times 4 \text{ KB} = 14.6 \text{ GB}$. Although the SUT has 16 GB of memory in total, often only 10 GB is available for buffer-cache. Ultimately, the high CPU utilization interfered with SNMP polling (as evidenced by a drop in network traffic) so the test was stopped.

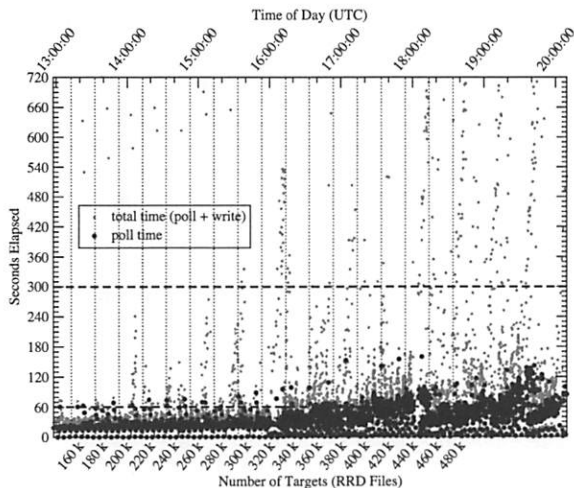


Figure 15: Scaling with fadvise: MRTG performance on the SUT progressively increasing to 483,000 targets and 53 MRTG daemons.

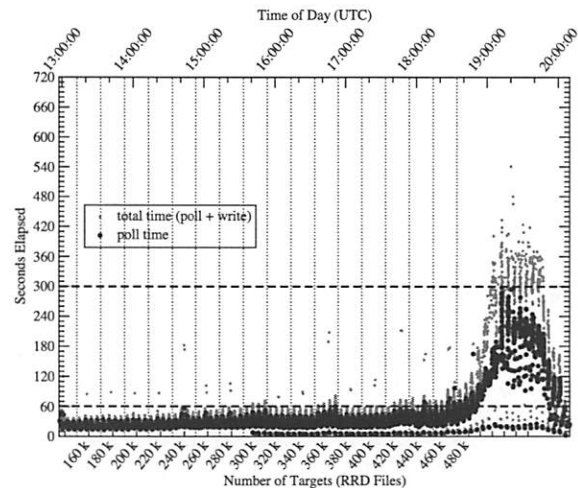


Figure 17: Scaling RRDCache with fadvise: MRTG performance on the SUT progressively increasing to 486,000 targets and 52 MRTG daemons.

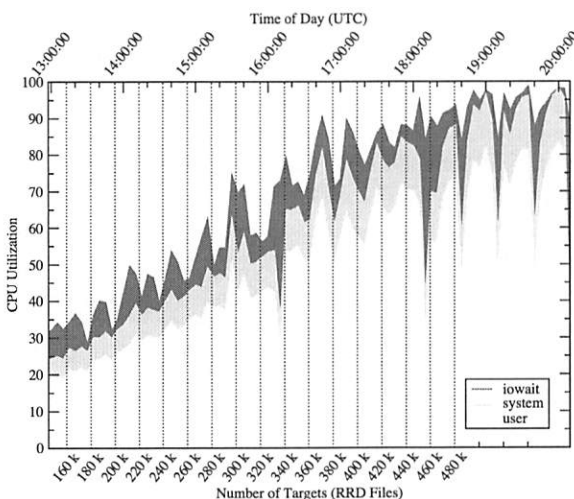


Figure 16: Scaling with fadvise: CPU utilization on the SUT while progressively increasing to 483,000 targets and 53 MRTG daemons.

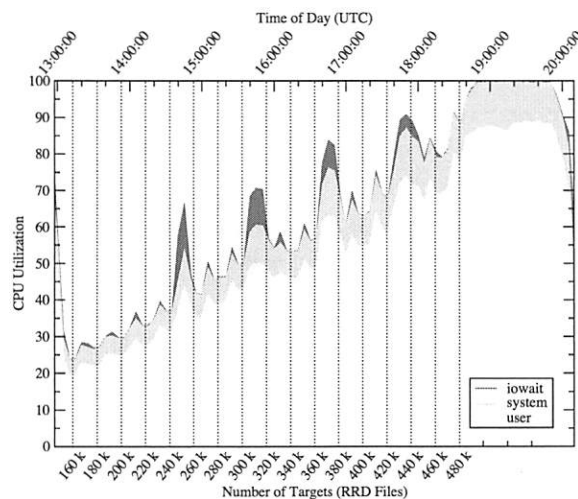


Figure 18: Scaling RRDCache with fadvise: CPU utilization on the SUT while progressively increasing to 486,000 targets and 52 MRTG daemons.

MRTG with fadvise RANDOM and RRDCache

Subsequently, we tested the scalability of MRTG using RRDCache combined with RRDTool patched to fadvise RANDOM. The performance results are shown in Figures 17 and 18. These combined techniques yielded the highest capacity, exceeding 400,000, but CPU utilization reached 100% and the RRDCache-writer could not complete its hourly updates within an hour, so an increasing backlog developed from which it didn't recover and the test was stopped.

Limitations

These scalability tests on our SUT show at least two capacity or performance limitations of large RRD and MRTG systems:

1. When buffer-cache is scarce, page fault rates peak at RRD aggregation times that are predictable offsets from zero hours UTC.

Synchronized aggregations, and thus consolidated data points with matching timestamps, are a convenience to RRDTool users when fetching or graphing the data. However, when updating RRD files in near real-time, there is clear performance consequence to synchronized aggregation because work is not distributed evenly across time.

2. CPU utilization approached or reached 100% when updating around 480,000 RRD files. This consists primarily of user mode CPU that we attribute to the MRTG and RRDCachewriter perl scripts. Thus, the next performance bottleneck limiting the scalability of MRTG systems is likely CPU.

We've shown that MRTG and RRD systems can scale to hundreds of thousands of targets or RRD files.

Without changing the RRD file read semantics, our `fadvise` `RANDOM` method allows us to scale to 320,000 target and files with acceptable performance on our system with 16 GB of memory. With slightly changed read semantics (because of the deferred RRD file updates), the `RRDCache` method scales higher. The factors that limit further scalability are (i) the CPU required for target processing (in perl) and (ii) `RRDTool`'s aggregations at synchronized times across all RRD files. Further gains can be achieved by profiling and optimizing the system software (e.g., `MRTG`) and by appropriately sizing the systems physical memory so that an even larger buffer-cache is available.

Related Work

Our work is informed by prior operating system and application performance improvement techniques.

Within the operating system, better buffer-cache management techniques can help reduce the number of disk reads and writes. There are a number of policies described in the literature (e.g., `FIFO`, `LRU`, `LFU`, `Clock`, `Random`, `Segmented FIFO`, `2Q` [10], and `LRU-K`). The readahead by the OS can limit the amount of useful data that can be cached. In some circumstances, improvements to the adaptive readahead algorithm can significantly improve performance [14].

The ability to accept hints or advice from applications with the aim of more efficiently managing resources and improving performance was implemented in the Pilot operating system [20]. An interface by which operating systems can accept such advice specifically to provide buffer management has long-since been suggested [25]. Later work finds that application "hints" to inform the operating system of file access patterns improves performance [15]. Today, some operating systems have support for application advice via the `fadvise` and `madvise` APIs [19].

A related approach is to change the kernel to include functionality which enables application guided buffer-cache control [2]. Another possibility is to simulate the cache replacement algorithm to build a reasonably accurate model of the contents of the cache for reordering reads and writes [4].

Within our version of the operating system (Linux 2.6), there are four I/O scheduling algorithms available: `Completely Fair Queuing (CFQ)` scheduling, `deadline elevator`, `NOOP scheduler` and `Anticipatory elevator scheduling` [21]. The scheduling algorithm can prioritize individual I/O requests, such as reads over writes, and therefore affects application performance when page faults occur.

A number of software systems inspired by the original `MRTG` have improved its performance in some ways. The current `MRTG`, `Cricket` [1], `Cacti` [5], and `Torrus` [23] applications use `RRDTool` [13] to achieve improved performance. `Cricket` [1] allows configuration of more parallel measurements per file,

but this offers only a modest performance improvement since tens to hundreds of thousands of RRD files would still be required. `RTG` [3] made significant changes in the polling (but that is not our bottleneck) and replaced the file I/O with relational database I/O. We have no reason to believe this would offer better I/O performance, and it significantly changes the user interface to the data. `JRobin` [11] completely reimplements `RRDTool` in java, improving performance in some areas but decreasing it in others and modifying the RRD file format in the process.

Recently, RRD users have proposed design changes or made customizations to introduce an application-level cache maintained by a daemon that intercepts updates [22, 12, 9].

Discussion and Future Work

Our investigation and experimentation thus far suggests at least the following potential items of future work.

- **File Types:** UNIX-based operating systems lack file types; a file is simply a stream of bytes and this is often cited as an advantage or, at least, a successful simplification. However, this is one reason that the RRD file update access pattern is not handled well by the adaptive readahead algorithm. Perhaps the readahead and other behaviors, such as caching, could be influenced or determined at file open time based on a file's type, as defined by file name extension (e.g., ".rrd") or by magic, the file command's magic number file.
- **File Read Performance:** Although we have disabled readahead to achieve better update performance, we have not thoroughly investigated its effect on RRD fetch or graphing (read) performance. We surmise that advising for random I/O helps read performance too, but have not carefully measured it. Also, we selected the Linux deadline I/O scheduler because it prioritizes reads over writes, but evaluating this decision is left for future work. Linux' `Completely Fair Queuing (CFQ)` scheduler may perform acceptably as well; we have not compared them.
- **RRD Update Interval:** Some network operators desire more frequent measurements, such as a one minute interval rather than five. Future work might explore if RRD scales similarly in this situation. We believe our page fault rate model is valid for all update intervals somewhat greater than that of the system's page replacement algorithm. (Note that dirty pages are flushed at five second intervals by `pdflush` in Linux 2.6.)

Our performance results suggest that the CPU load would be a limiting factor as the update interval decreases, i.e., if updates are more frequent. Perhaps simply choosing a one minute

interval would constrain the capacity to about one fifth that of when using a five minute interval.

Judicious partitioning of the set of targets would help, e.g., using a one minute interval for measuring the core and/or distribution links and a five minute interval for more numerous access ports.

- **MRTG CPU Utilization:** As is to be expected in system performance work, we found that eliminating the I/O bottleneck exposed the next bottleneck, CPU, that limits scalability. It seems this high CPU utilization is primarily due to the MRTG perl script, thus profiling and optimizing it could improve performance.
- **“Gaming” the Readahead Algorithm:** While our platform provides the BSD and POSIX *advise* APIs, others do not or do not yet completely implement them. Can we instead “game” their readahead algorithms, for instance by performing otherwise unnecessary no-op seek operations, to likewise disable readahead? What is the performance cost of doing so? If adaptive readahead algorithms are similar, this might also have portability benefits. (We’ve observed that some operating systems use a readahead of zero initially,⁸ so they exhibit the desired behavior for RRD files without needing advice or adaptation.)
- **RRD File Design:** We’ve seen how the RRD file organization influences its update performance. Is there a better organization for RRD files? For example, locality of data for updates would improve if RRAs with the same number of PDPs (but different consolidation functions) could be interleaved in the same block so that their corresponding data points are nearby when updated.

Is there a way to avoid synchronized aggregations or consolidations across all RRD files? Perhaps we can introduce a stochastic component to skew those updates slightly in time. This is difficult to do without affecting RRD file read semantics and without introducing an independent thread to perform updates.

Conclusions

In conclusion, we’ve provided a general analysis method and two new tools, *fincore* (available at [18]) and *fadvise* (available at [16]), that expose readahead and buffer-cache behaviors in running systems. Without such tools, these performance-critical aspects of

⁸Apple’s OS X with HFS+ file-system has an initial readahead of zero.

the operating system are hidden from system administrators and users.

By both modeling and simulation, we’ve provided a detailed analysis of the I/O characteristics of RRD file updates. We’ve shown how the locality of RRD file accesses can be leveraged, limiting page faults and disk I/O, resulting in improved performance and scalability for RRD systems. We’ve found that RRD buffer-cache utilization and page faults are defined by subtleties in the RRD file format and RRD-Tool’s access pattern, rather than simply being defined by file size. This is advantageous because it means that larger RRD systems can be operated than would otherwise be thought.

We’ve outlined two effective methods to improve RRD performance. The first, *RRDCache* (available at [6]), is what we’ve called *application-level caching or buffering*. The second, for which we provide a patch to RRDTool (available at [17]), issues *application advice* to the operating system to select readahead and buffer-cache behavior appropriate for random RRD file I/O. While the two methods are starkly different, both eliminate the buffer-cache memory bottleneck that has been observed in large RRD network measurement systems. Conservatively, either technique *triples* the capacity of such systems. Together, these *complementary* techniques can be applied to maximize performance.

Finally, we’ve shown that system tuning and minor capacity-enhancing code changes improve Round Robin Database performance so that RRDTool can be used for even the largest managed networks.

Acknowledgments

We thank Hideko Mills for her support, Michael Swift for his valuable input, and Robert Plankers and Kevin Kettner II for their assistance with system administration. Mark Plaksin provided helpful review feedback.

kSar [7] proved useful for conveniently visualizing sar data.

This work is a collaboration with the University of Wisconsin’s Division of Information Technology.

Author Biographies

David Plonka is a graduate student and research assistant in Computer Sciences at the University of Wisconsin-Madison. He received a B.S. from Carroll College in Waukesha, Wisconsin in 1991. He can be reached at plonka@cs.wisc.edu.

Archit Gupta is a graduate student in Computer Sciences at the University of Wisconsin-Madison. His primary interest areas are Systems and Networking. He can be reached at archit@cs.wisc.edu.

Dale Carder is a senior network engineer for the University of Wisconsin-Madison and WiscNet. He can be reached at dwcarder@doit.wisc.edu.

Bibliography

- [1] Allen, J. R., "Driving via the Rear-View Mirror: Managing a Network with Cricket," *Conference on Network Administration*, pp. 1-10, 1999.
- [2] Arpaci-Dusseau, A. C., R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici, "Transforming Policies into Mechanisms with Infokernel," *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October, 2003.
- [3] Beverly, R., "RTG: A Scalable SNMP Statistics Architecture for Service Providers," *Proceedings of the 16th Conference on Systems Administration (LISA 2002)*, Philadelphia, PA, pp. 167-174, November 3-8, 2002.
- [4] Burnett, N. C., J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Exploiting Gray-Box Knowledge of Buffer-Cache Contents," *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pp. 29-44, Monterey, CA, June, 2002.
- [5] *Cacti*, <http://www.cacti.net>.
- [6] Carder, D., *RRDCache*, <http://net.doit.wisc.edu/~dwcarder/rrdcache/>.
- [7] Cherif, X., *kSar*, <http://ksar.atomique.net>.
- [8] Gorman, M., *Understanding the Linux Virtual Memory Manager*, Prentice Hall, 2004.
- [9] Hustace, D., *Queueing RRD*, http://www.opennms.org/index.php/Queueing_RRD.
- [10] Johnson, T. and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB '94, Proceedings of 20th International Conference on Very Large Data Bases*, Santiago de Chile, Chile, pp. 439-450, September 12-15, 1994.
- [11] Markovic, S. and A. Vandamme, *JRobin*, <http://www.jrobin.org>.
- [12] Oetiker, T., *RRD Accelerator Design proposal*, <http://oss.oetiker.ch/rrdtool-trac/wiki/RRDaccelerator>.
- [13] Oetiker, T., "MRTG: The Multi Router Traffic Grapher," *Proceedings of the 12th Conference on Systems Administration (LISA-98)*, Boston, MA, USA, pp. 141-148, December 6-11, 1998.
- [14] Pai, R., B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement Through Readahead Optimization," *Proceedings of the Linux Symposium*, 2004.
- [15] Patterson, R. H., G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *SOSP*, pp. 79-95, 1995.
- [16] Plonka, D., *The fadvise command*, <http://net.doit.wisc.edu/~plonka/fadvise/>.
- [17] Plonka, D., *The fadvise random patch to RRDTool*, http://net.doit.wisc.edu/~plonka/rrdtool_fadvise/.
- [18] Plonka, D., *The fincore command*, <http://net.doit.wisc.edu/~plonka/fincore/>.
- [19] *fadvise*, http://www.opengroup.org/onlinepubs/009695399/functions/posix_fadvise.html.
- [20] Redell, D. D., Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, Vol. 23, Num. 2, pp. 81-92, 1980.
- [21] Shakshober, D. J., *Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel*, <http://www.redhat.com/magazine/008jun05/features/schedulers/>.
- [22] Simonet, P., *Post to rrd-developers mailing list*.
- [23] Sinyagin, S., *Torrus: The Data Series Processing Framework*, <http://torrus.org>.
- [24] Snyder, P., "tmpfs: A Virtual Memory File System," *Proceedings of the Autumn 1990 EUUG Conference*, Nice, France, pp. 241-248, 1990.
- [25] Stonebraker, M., "Operating System Support for Database Management," *Communications of the ACM*, Vol. 24, Num. 7, pp. 412-418, 1981.

Appendix: Performance Recommendations for RRD and MRTG Systems

- When building a very large RRD measurement system, dedicate the machine to this purpose. Since RRD is a file-based database, it relies on the buffer-cache that is shared across all system activity. Because of RRD's unique file access characteristics and buffering requirements, it is easier to achieve performance gains by tuning the system just for RRD.
- Use an RRDTool that has our `advise RANDOM` patch. On systems that have a fairly aggressive initial readahead (such as Linux), this will very likely increase file update performance by reducing the page fault rate and the buffer-cache memory required.
- Avoid file-level backups of RRD files unless the set of RRD files completely fit into buffer-cache memory. File-level backups read each modified file completely and sequentially; this can fill the buffer-cache and subsequently causes more page faults on RRD updates. Backups are essentially indistinguishable from application access, and thus unnecessarily populate the system's buffer-cache with content that won't be re-used soon. (Note that backup programs could call `advise NOREUSE` or `advise DONTNEED` to inform the operating system that the file content will not be re-used.)
- Split MRTG targets into a number of groups and run a separate daemon for each. In our system, we reconfigure daily and run a `target_splitter` script to produce a new set of ".cfg" files each with approximately 10,000 targets per MRTG daemon. Note that polling performance is also influenced by the SNMP agent performance on the network device polled. So, if the splitting results in grouping like targets together based on the model of device monitored, there could be quite a disparity in time to complete the MRTG "poll targets" phase.
- Do not create RRD files all at once. By staggering the start times, updates to like RRAs will cross block boundaries at different times, distributing the page faults that occur on block boundary crossings. As a network is deployed and grows, these RRD file start times would naturally be staggered, but this could be quite different when introducing measurement to an existing deployed network.
- Run a caching resolver or a nameserver on the localhost, i.e., the MRTG system itself. This reduces "poll targets" latency due to host name resolution; MRTG performs very many DNS name resolutions when hostnames are used (rather than IP addresses) in target definitions.
- Configure an appropriate number of forks for each MRTG daemon to minimize the time for the "poll targets" phase. On our system, 4 forks per daemon works well to keep polling in the tens of seconds for 10,000 targets. This might differ for a wide-area network.
- Place RRD files in a file-system of their own, ideally one associated with separate logical volumes or disks. This gives the system administrator flexibility to change mount options or other file-system options. It also isolates the system activity data (e.g., as displayed by `sar`) from unrelated activity.
- Consider mounting the file-system that contains the RRD files with the "noatime" and "nodiratime" options so that RRD file reads do not require an update to the file inode block. Of course the effect of this is that file access times will be inaccurate, but often these are not of interest for ".rrd" files.
- Consider enabling `dir_index` on ext file-systems to speed up lookups in large directories. MRTG places all RRD files in the same directory, and we've scaled to hundreds of thousands.

Stork: Package Management for Distributed VM Environments

Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John H. Hartman – University of Arizona

ABSTRACT

In virtual machine environments each application is often run in its own virtual machine (VM), isolating it from other applications running on the same physical machine. Contention for memory, disk space, and network bandwidth among virtual machines, coupled with an inability to share due to the isolation virtual machines provide, leads to heavy resource utilization. Additionally, VMs increase management overhead as each is essentially a separate system.

Stork is a package management tool for virtual machine environments that is designed to alleviate these problems. Stork securely and efficiently downloads packages to physical machines and shares packages between VMs. Disk space and memory requirements are reduced because shared files, such as libraries and binaries, require only one persistent copy per physical machine. Experiments show that Stork reduces the disk space required to install additional copies of a package by over an order of magnitude, and memory by about 50%. Stork downloads each package once per physical machine no matter how many VMs install it. The transfer protocols used during download improve elapsed time by 7X and reduce repository traffic by an order of magnitude. Stork users can manage groups of VMs with the ease of managing a single machine – even groups that consist of machines distributed around the world. Stork is a real service that has run on PlanetLab for over four years and has managed thousands of VMs.

Introduction

The growing popularity of virtual machine (VM) environments such as Xen [3], VMWare [31], and Vservers [17, 18], has placed new demands on package management systems (e.g., apt [2], yum [36], RPM [27]). Traditionally, package management systems deal with installing and maintaining software on a single machine whether virtual or physical. There are no provisions for inter-VM sharing, so that multiple VMs on the same physical machine individually download and maintain separate copies of the same package. There are also no provisions for inter-machine package management, centralized administration of which packages should be installed on which machines, or allowing multiple machines to download the same package efficiently. Finally, current package management systems have relatively inflexible security mechanisms that are either based on implicit trust of the repository, or public/private key signatures on individual packages.

Stork is a package management system designed for distributed VM environments. Stork has several advantages over existing package management systems: it provides *secure* and *efficient* inter-VM package sharing on the same physical machine; it provides centralized package management that allows users to determine which packages should be installed on which VMs without configuring each VM individually; it allows multiple physical machines to download the same package efficiently; it ensures that package

updates are propagated to the VMs in a timely fashion; and it provides a flexible security mechanism that allows users to specify which packages they trust as well as delegate that decision on a per-package basis to other (trusted) users.

Stork's inter-VM sharing facility is important for reducing resource consumption caused by package management in VM environments. VMs are excellent for isolation, but this very isolation can increase the disk, memory, and network bandwidth requirements of package management. It is very inefficient to have each VM install its own copy of each package's files. The same is true of memory: if each VM has its own copy of a package's files then it will have its own copy of the executable files in memory. Memory is often more of a limiting factor than disk, so Stork's ability to share package files between VMs is particularly important for increasing the number of VMs a single physical machine can support. In addition, Stork reduces network traffic by only downloading a package to a physical machine once, even if multiple VMs on the physical machine install it.

Stork's inter-machine package management facility enables centralized package management and efficient, reliable, and timely package downloads. Stork provides package management utilities and configuration files that allow the user to specify which packages are to be installed on which VMs. Machines download packages using efficient transfer mechanisms such as BitTorrent [9] and CoBlitz [22], making downloads

efficient and reducing the load on the repository. Stork uses fail-over mechanisms to improve the reliability of downloads, even if the underlying content distribution systems fail. Stork also makes use of publish/subscribe technology to ensure that VMs are notified of package updates in a timely fashion.

Stork provides all of these performance benefits without compromising security; in fact, Stork has additional security benefits over existing package management systems. First, Stork shares files securely between VMs. Although a VM can delete its link to a file, it cannot modify the file itself. Second, a user can securely specify which packages he or she trusts and may delegate this decision for a subset of packages to another user. Users may also trust other users to know which packages *not* to install, such as those with security holes. Each VM makes package installation decisions based on a user's trust assumptions and will not install packages that are not trusted. While this paper touches on the security aspects of the system that are necessary to understand the design, a more rigorous and detailed analysis of security is available through documentation on our website [29].

In addition, Stork is flexible and modular, allowing the same Stork code base to run on a desktop PC, a Vserver-based virtual environment, and a PlanetLab node. This is achieved via pluggable modules that isolate the platform-specific functionality. Stork accesses these modules through a well-defined API. This approach makes it easy to port Stork to different environments and allows the flexibility of different implementations for common operations such as file retrieval.

Stork has managed many thousands of VMs and has been deployed on PlanetLab [23, 24] for over four years. Stork is currently running on hundreds of PlanetLab nodes and its package repository receives a request roughly every ten seconds. Packages installed in multiple

VMs by Stork typically use over an order of magnitude less space and 50% the memory of packages installed by other tools. Stork also reduces the repository load by over an order of magnitude compared to HTTP-based tools. Stork is also used in the Vserver [18] environment and can also be used in non-VM environments (such as on a home system) as an efficient and secure package installation system. The source code for Stork is available at <http://www.cs.arizona.edu/stork>.

Stork

Stork provides manual management of packages on individual VMs using command-line tools that have a syntax similar to `apt` [2] or `yum` [36]. Stork also provides centralized management of groups of VMs. This section describes an example involving package management, the configuration files needed to manage VMs with Stork, and the primary components of Stork.

An Example

Consider a system administrator that manages thousands of machines at several sites around the globe. The company's servers run VM software that allow different production groups more flexible use of the hardware resources. In addition, the company's employees have desktop machines that have different software installed depending on their use.

The system administrator has just finished testing a new security release for a fictional package `foobar` and she decides to have all of the desktop machines used for development update to the latest version along with any testing VMs that are used by the coding group. The administrator modifies a few files on her local machine, signs them using her private key, and uploads them to a repository. Within minutes all of the desired machines that are online have the updated `foobar` package installed. As offline machines come online or new VMs

File Type	Repository	Client	Central Mgmt	Signed and Embedded
User Private Key	No	No	Yes	No
User Public Key	No †	Yes	Yes	No
Master Configuration File	No †	Yes	Yes	No
Trusted Packages (TP)	Yes	Yes	Yes	Yes
Pacman Packages	Yes	No	Yes	Yes
Pacman Groups	Yes	No	Yes	Yes
Packages (RPM, tar.gz)	Yes	Yes	Yes	Secure Hash
Package Metadata	Yes	Yes	Yes	No
Repository Metahash	Yes	Yes	No	Signed Only

Table 1: Stork File Types: This table shows the different types of files used by Stork. The repository column indicates whether or not the file is obtained from the repository by the clients. The client column indicates whether or not the file is used for installing packages or determining which packages should be installed locally based upon the files provided by the centralized management system. The centralized management column indicates if the files are created by the management tools. The signed/embed column indicates which files are signed and have a public key embedded in their name.

† In order to automatically deploy Stork on PlanetLab, this restriction is relaxed. See the PlanetLab section for more details.

are created, they automatically update their copies of foobar as instructed.

The subsequent sections describe the mechanisms Stork uses to provide this functionality to its users. The walkthrough section revisits this example and explains in detail how Stork provides the functionality described in this scenario.

File Types

Stork uses several types of files that contain different information and are protected in different ways (Table 1). The user creates a public/private key pair that authenticates the user to the VMs he or she controls. The *public key* is distributed to all of the VMs and the *private key* is used to sign the configuration files. In our previous example, the administrator's public key is distributed to all of the VMs under her control. When files signed by her private key were added to the repository, the authenticity of these files was independently verified by each VM using the public key.

The *master configuration file* is similar to those found in other package management tools and indicates things such as the transfer method, repository name, user name, etc. It also indicates the location of the public key that should be used to verify signatures.

The user's trusted packages file (*TP file*) indicates which packages the user considers valid. The TP file does not cause those packages to be installed, but instead indicates trust that the packages have valid contents and are candidates for installation. For example, while the administrator was testing the latest release of foobar she could add it to her trusted packages file because she believes the file is valid.

There are two pacman files used for centralized management. The *groups.pacman* file allows VMs to be categorized into convenient groups. For example, the administrator could configure her pacman groups file to create separate groups for VMs that perform different tasks. VMs can belong in multiple groups such as ALPHA and ACCOUNTING for an alpha test version of accounting software. Any package management instructions for either the ALPHA group or the ACCOUNTING group would be followed by this VM.

The *packages.pacman* file specifies what actions should be done on a VM or a group of VMs. Packages can be installed, updated, or removed. Installation is different from updating in that installation will do nothing if there is a package that meets the criteria already installed while update ensures that the preferred version of the package is installed. For example, when asked to install foobar, if any version of the package is currently installed then no operation will occur. If asked to update foobar, Stork checks to see if the administrator's TP file specifies a different version of foobar and if so, replaces the current version with the new version.

The *packages* (for example, the foobar RPM itself) contain the software that is of interest to the user. The *package metadata* is extracted from packages and is published by the repository to describe the packages that are available. The *repository metahash* is a special file that is provided by the repository to indicate the current repository state.

Architecture

Stork consists of four main components:

- a *repository* that stores configuration files, packages, and associated metadata;
- a set of *client tools* that are used in each Stork client VM to manage its packages by interacting either directly with the repository or through the nest when it is available;
- a *nest* process that runs on physical machines and coordinates sharing between VMs as well as providing repository metadata updates to its client VMs and downloading packages;
- and *centralized management tools* that allows a user to control many VMs concurrently, create and sign packages, upload packages to the repository, etc.

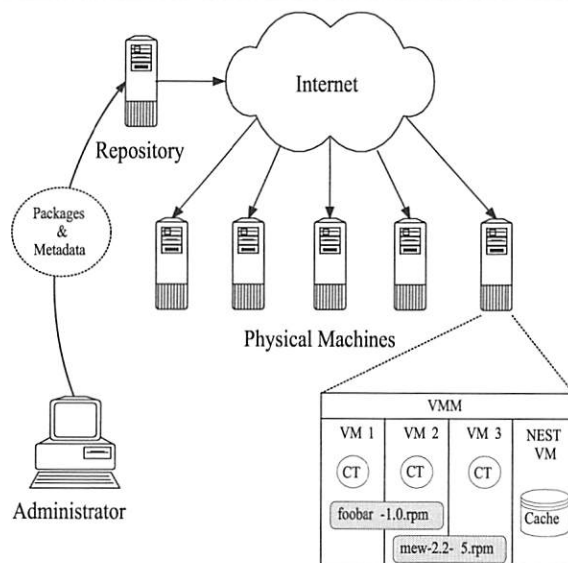


Figure 1: Stork Overview. Stork allows centralized administration and sharing of packages. The administrator publishes packages and metadata on the repository. Updates are propagated to VMs running on distributed physical machines. Each physical machine contains a single nest VM, and one or more client VMs that run the Stork client tools.

The client tools consist of the *stork command-line tool* (referred to simply as *stork*), which allows users to install packages manually, and *pacman*, which supports centralized administration and automated package installation and upgrade. While a client VM may communicate with the repository directly, it is far more

efficient for client VMs to interact with their local nest process, who interacts with the repository on their behalf.

Repository

The Stork repository's main task is to serve files much like a normal web server. However, the repository is optimized to efficiently provide packages to Stork client VMs. First, the repository provides secure user upload of packages, trusted packages files, and pacman packages and groups files. Second, the repository pushes notifications of new content to interested VMs. Third, the repository makes packages available via different efficient transfer mechanisms such as BitTorrent.

Handling Uploaded Data The Stork repository allows multiple users to upload files while retaining security. TP, groups.pacman, and packages.pacman files must be signed by the user that uploads them. Every signed file has a timestamp for the signature embedded in the portion of the file protected by the signature. The public key of the user is embedded in the file name of the signed file (similar to self-certifying path names [19]). This avoids naming conflicts and allows the repository to verify the signature of an uploaded file. The repository will only store a signed file with a valid signature that is newer than any existing signed file of the same name. This prevents replay attacks and allows clients to request files that match a public key directly.

Packages and package metadata are treated differently than configuration files. These files are not signed, but instead incorporate a secure hash of their contents in their names. This prevents name collisions and allows clients to request packages directly by secure hash. In all cases, the integrity of a file is verified by the recipient before it is used (either by checking the signature or the secure hash, as appropriate). The repository only performs these checks itself to prevent pollution of the repository and unnecessary downloads, rather than to ensure security on the clients.

Pushing Notifications The repository notifies interested clients when the repository contents have changed. The repository provides this functionality by pushing an updated repository metahash whenever data has been added to the repository. However, this does not address the important question of *what* data has been updated. This is especially difficult to address when VMs may miss messages or suffer other failures.

One solution is for the repository to push out hashes of all files on the repository. As there are many thousands of metadata files on the repository, it is too costly to publish the individual hashes of all of them and have the client VMs download each metadata file separately. Instead, the repository groups metadata files together in a tarball organized by type. For example, one tarball contains all of the trusted packages files, another with all of the pacman files, etc. The hashes of these tarballs are put into the repository

metahash which is pushed to each interested client VM. No matter how many updates the client VM misses, it can examine the hash of the local tarballs and the hashes provided by the repository and determine what needs to be retrieved.

Efficient Transfers The repository makes all of its files available for download through HTTP. However, having each client download its files via separate HTTP connections is prohibitively expensive. The repository therefore supports different transfer mechanisms for better scalability, efficiency, and performance. Some transfer mechanisms are simple (like CoBlitz and Coral) which require no special handling by the repository and others (like BitTorrent) which do.

To support BitTorrent [9] downloads the repository runs a BitTorrent tracker and a modified version of the btlanchmany daemon provided by BitTorrent. The btlanchmany daemon monitors a directory for any new or updated files. When a new file is uploaded to the repository it is placed in the monitored directory. When the daemon notices the new file it creates a torrent file that is later seeded. Unique naming is achieved by appending the computed hash of the shared file to the name of the torrent. The torrent file is placed in a public location on the repository for subsequent download by the clients through HTTP.

Client Tools

The client tools are used to manage packages in a client VM and include the stork, pacman, and stork_receive_update commands. The stork tool uses command-line arguments to install, update, and remove packages. Its syntax is similar to apt [2] or yum [36]. The stork tool resolves dependencies and installs additional packages as necessary. It also upgrades and removes packages. The stork tool downloads the latest metadata from package repositories, verifies that packages are trusted by the user's TP file, and only installs trusted files.

Package management with the stork tool is a complex process involving multiple steps including *dependency resolution*, *trust verification*, *download*, and *installation*. For example, consider the installation of the foobar package. Assume foobar depends on a few other packages, such as emacs and glibc, before foobar itself can be installed. In order to perform the installation of foobar, the stork tool must determine whether foobar, emacs, and glibc are already installed on the client and if not, locate candidate versions that satisfy the dependencies. These steps are similar to those performed by other package managers [2, 36, 27]. Finally Stork ensures that those candidates satisfy the trust requirements that the user has specified.

Figure 2 shows a TP file example. This file specifically allows emacs-2.2-5.i386.rpm, several versions of foobar, and customapp-1.0.tar.gz to be installed. Each package listed in the TP file includes the hash of the package, and only packages that match the hashes

may be installed. It trusts the planetlab-v4 user to know the validity of any package it says (this user has a list of hashes of all of the Fedora Core 4 packages). It also trusts the stork user to know the validity of any packages that start with "stork".

Once satisfactory trusted candidates have been found, Stork downloads the packages from the repository and verifies that the packages it downloaded match the entries in the TP file, including the secure hashes. Finally, the packages themselves are installed.

Package removal is much less complex than installation. Before removing a package, the stork command first checks to see if other packages depend upon the package to be removed. For RPM packages, stork leverages the rpm command and its internal database to check dependencies. Tar packages do not support dependencies at this time and can always be removed. If there are dependencies that would be broken by removal of the package, then stork reports the conflict and exits. Stork removes an installed package by deleting the package's files and running the uninstall scripts for the package.

The pacman ("package manager") tool is the entity in a VM that locally enacts centralized administration decisions. The pacman tool invokes the appropriate stork commands based on two configuration files: groups.pacman (Figure 3) and packages.pacman (Figure 4). The groups.pacman file is optional and defines VM groups that can be used by an administrator to manage a set of VMs collectively. The groups.pacman syntax supports basic set operations such as union, intersection, compliment, and difference. For example, an administrator for a service may break their VMs into alpha VMs, beta VMs, and production VMs. This

allows developers to test a new release on alpha VMs (where there are perhaps only internal users) before moving it to the beta VMs group (with beta testers) and finally the the production servers.

```
<GROUPS>
<GROUP NAME="ALPHA">
<INCLUDE NAME="planetlab1.arizona.net"/>
<INCLUDE NAME="planetlab2.arizona.net"/>
</GROUP>
<GROUP NAME="ACCOUNTING">
<INCLUDE NAME="ALPHA"/>
<INCLUDE NAME="p11.unm.edu"/>
</GROUP>
</GROUPS>
```

Figure 3: Example groups.pacman. The "ALPHA" group consists of two machines in Arizona. The "ACCOUNTING" group also includes a machine at the University of New Mexico.

The packages.pacman file specifies which packages should be installed, updated, or removed in the current VM based on a combination of VM name, group, and physical machine. This makes it easy, for example, to specify that a particular package should be installed on all VMs on a physical machine, while another package should only be installed on alpha VMs, etc.

Although pacman can be run manually, typically it is run automatically via one of several mechanisms. First, pacman establishes a connection to the stork_receive_update daemon. This daemon receives the repository metahashes that are pushed by the repository whenever there is an update. Upon receiving this notification, stork_receive_update alerts pacman to the new

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>
<!-- Trust some packages that the user specifically allows -->
<FILE PATTERN="emacs-2.2-5.i386.rpm" HASH="aed4959915ad09a2b02f384d140c4\
626b0eba732" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.01.i386.rpm" HASH="16b6d22332963d54e0a034c11376a\
2066005c470" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.0.i386.rpm" HASH="3945fd48567738a28374c3b238473\
09634ee37fd" ACTION="ALLOW"/>
<FILE PATTERN="simple-1.0.tar.gz" HASH="23434850ba2934c39485d293403e3\
293510fd341" ACTION="ALLOW"/>
<!-- Allow access to the planetlab Fedora Core 4 packages -->
<USER PATTERN="*" USERNAME="planetlab-v4" PUBLICKEY="MFwwDQYJKoZIhvcNAQEB\
BQADSwAwSAJBALtGteQpDLa0kYv+k1FWTk1H9Y7frYh15JV1hgJa5PlGI3yK+R22UsD65_J4P\
V92RUgVd_uJMuB8Q4bilw4o6JMCAwEAAQ" ACTION="ALLOW"/>
<!-- Allowing the 'stork' user lets stork packages be installed -->
<USER PATTERN="stork*" USERNAME="stork" PUBLICKEY="MFwwDQYJKoZIhvcNAQEBBQADSwAw\
SAJBAKgZCjFKD19ISoclfBuZsQze6bXtu+QYF64TLQ1I9fgEg2CDyGQVOsZ2CaX1ZEZ_069AYZ\
p8nj+YJLIJM3+W3DMCAwEAAQ" ACTION="ALLOW"/>
</TRUSTEDPACKAGES>
```

Figure 2: Example TP File. This file specifies what packages and users are trusted. Only packages allowed by a TP file may be installed. FILE actions are used to trust individual packages. USER actions allow hierarchical trust by specifying a user whose TP file is included. The signature, timestamp, and duration are not shown and are contained in an XML layer that encapsulates this file.

information. A change to the repository metahash indicates that the repository contents have changed which in turn may change which packages are installed, etc. Second, when `stork_receive_update` is unavailable `pacman` wakes up every 5 minutes and polls the repository for the repository metahash. As before, if there is a discrepancy between the stored data and the described data, `pacman` downloads the updated files. Third, `pacman` also runs when its configuration files change.

The `stork_receive_update` daemon runs in each client VM and keeps the repository's metahash up-to-date. Metadata is received from the repositories using both push and pull. Pushing is the preferred method because it reduces server load, and is accomplished using a multicast tree or publish/subscribe system such as PsEPR [5]. Heartbeats are pushed if no new metahash is available. If `stork_receive_update` doesn't receive a regular heartbeat it polls the repository and downloads new repository metahash if necessary. This download is accomplished using an efficient transfer mechanism from one of Stork's *transfer modules* (discussed further in the transfer modules section). This combination of push and pull provides an efficient, scalable, fault tolerant way of keeping repository information up-to-date in the VMs.

Nest

The Stork nest process enables secure file-sharing between VMs, prevents multiple downloads of the same content by different VMs, and maintains up-to-date repository metadata. It accomplishes these in two ways. First, it operates as a shared cache for its client VMs, allowing metadata and packages to be downloaded once and used by many VMs. Second, it performs package installation on behalf of the VMs, securely sharing read-only package files between multiple VMs that install the package (discussed further in the sharing section). The nest functionality is implemented by the `stork_nest` daemon.

The `stork_nest` daemon is responsible for maintaining connections with its client VMs and processing requests that arrive over those connections (typically via a socket, although this is configurable). A client must first authenticate itself to `stork_nest`. The authentication persists for as long as the connection is established. Once authenticated, the daemon then fields

requests for file transfer and sharing. File transfer operations use the shared cache feature of the repository to provide cached copies of files to the clients. Sharing operations allow the clients to share the contents of packages using the *prepare interface* (discussed further in the section on prepare modules).

Typically, the nest runs on each machine that runs Stork; however, there may be cases where the nest is not run, such as in a desktop machine or a server that does not use VMs. In the case where no nest is running or the nest process fails, the client tools communicate directly with the repository.

Centralized Management Tools

The centralized management tools allow Stork users to manage their VMs without needing to contact the VMs directly. In our example the administrator wanted to install `foobar` automatically on applicable systems under her control rather than logging into them individually. Unlike the client tools that are run in Stork client VMs, the centralized management tools are typically run on the user's desktop machine. They are used to create TP files, `pacman` packages and groups files, the master configuration file, public/private keypairs, etc. These files are used by the client tools to decide what actions to perform on the VM. In addition to managing these files, the centralized management tools also upload metadata and/or packages to the repository, and assist the user in building packages.

The main tool used for centralized management is `storkutil`, a command-line tool that has many different functions including creating public/private key pairs, signing files, extracting metadata from packages, and editing trusted packages, `pacman` packages and groups files. Administrators use this tool to create and modify the files that control the systems under their control. While files can be edited by other tools and then resigned, `storkutil` has the advantage of automatically resigning updated files. After updating these files they are then uploaded to the repository.

Stork on PlanetLab

Stork currently supports the Vserver environment, non-VM machines, and PlanetLab [23, 24]. The PlanetLab environment is significantly different from

```
<PACKAGES>
  <CONFIG SLICE="stork" GROUP="ACCOUNTING">
    <INSTALL PACKAGE="foobar" VERSION="2.2"/>
    <REMOVE PACKAGE="vi"/>
  </CONFIG>
  <CONFIG>
    <UPDATE PACKAGE = "firefox"/>
  </CONFIG>
</PACKAGES>
```

Figure 4: Example packages.pacman. VMs in the slice (a term used to mean a VM on PlanetLab) “stork” and in the group “ACCOUNTING” will have `foobar` 2.2 installed and `vi` removed. All VMs in this user's control will have `firefox` installed and kept up-to-date with the newest version.

the other two, so several extensions to Stork have been provided to better support it.

PlanetLab Overview

PlanetLab consists of over 750 nodes spread around the world that are used for distributed system and network research. Each PlanetLab node runs a custom kernel that superficially resembles the Vserver [18] version of Linux. However there are many isolation, performance, and functionality differences.

The common management unit in PlanetLab is the *slice*, which is a collection of VMs on different nodes that allow the same user(s) to control them. A node typically contains many different VMs from many different slices, and slices typically span many different nodes. The common PlanetLab (mis)usage of the word “slice” means both the collection of similarly managed VMs and an individual VM.

Typical usage patterns on PlanetLab consist of an authorized user creating a new slice and then adding it to one or more nodes. Many slices are used for relatively short periods of time (a week or two) and then removed from nodes (which tears down the VMs on those nodes). It is not uncommon for a group that wants to run an experiment to create and delete a slice that spans hundreds of nodes in the same day. There are relatively loose restrictions as to the number of nodes slices may use and the types of slices that a node may run so it is not uncommon for slices to span all PlanetLab nodes.

Bootstrapping Slices on PlanetLab

New slices on PlanetLab do not have the Stork client tools installed. Since slices are often short-lived and span many nodes, requiring the user to log in and install the Stork client tools on every node in a slice is impractical. Stork makes use of a special *initscript* to automatically install the Stork client tools in a slice. The *initscript* is run whenever the VMM software instantiates a VM for the slice on a node. The Stork *initscript* communicates with the nest on the node and asks the nest to share the Stork client tools with it. If the nest process is not working, the *initscript* instead retrieves the relevant RPMs securely from the Stork repository.

Centralized Management

Once the Stork client tools are running they need the master configuration file and public key for the slice. Unfortunately the ssh keys that are used by PlanetLab to control slice access are not visible within the slice, so Stork needs to obtain the keys through a different mechanism. Even if the PlanetLab keys were available it is difficult to know which key to use because many users may be able to access the same VM. Even worse, often a different user may want to take control of a slice that was previously managed by another user. Stork’s solution is to store the public key and master configuration file on the Stork repository.

The repository uses PlanetLab Central’s API to validate that users have access to the slices they claim and stores the files in a area accessible by https. The client tools come with the certificate for the Stork repository which *pacman* and *stork* use to securely download the public key and master configuration file for the slice. This allows users to change the master configuration file or public key on all nodes by simply adding the appropriate file to the Stork repository.

Modularity

Stork is highly modular and uses several interfaces that allow its functionality to be extended to accommodate new protocols and package types:

Transfer A transfer module implements a transport protocol. It is responsible for retrieving a particular object given the identifier for that object. Transfer protocols currently supported by Stork include Co-Blitz [21], BitTorrent [9], Coral [12], HTTP, and FTP.

Share A share module is used by the Stork nest to share files between VMs. It protects files from modification, maps content between slices, and authenticates client slices. Currently Stork supports PlanetLab and Linux VServers. Using an extensible interface allows Stork to be customized to support new VM environments.

Package A package module provides routines that the Stork client tools use to install, remove, and interact with packages. It understands several package formats (RPM, tar) and how to install them in the current system.

Prepare A prepare module prepares packages for sharing. Preparing a package typically involves extracting the files from the package. The Prepare interface differs from the Package interface in that package install scripts are not run and databases (such as the RPM database) are not updated. The nest process uses the prepare module to ready the package files for sharing.

Transfer Modules

Transfer modules are used to download files from the Stork repository. Transfer modules encapsulate the necessary functionality of a particular transfer protocol without having to involve the remainder of Stork with the details.

Each transfer module implements a *retrieve_files* function that takes several parameters including the name of the repository, source directory on the repository, a list of files, and a target directory to place the files in. The transfer module is responsible for opening and managing any connections that it requires to the repositories. A successful call to *retrieve_files* returns a list of the files that were successfully retrieved.

Transfer modules are specified to Stork via an ordered list in the main Stork configuration file. Stork always starts by trying the first transfer module in the

list. If this transfer module should fail or return a file that is old, then Stork moves on to the next module in the list.

Content Retrieval Modules

CoBlitz uses a content distribution network (CDN) called CoDeeN [33] to support large files transfers without modifying the client or server. Each node in the CDN runs a service that is responsible for splitting large files into chunks and reassembling them. This approach not only reduces infrastructure and the need for resource provisioning between services, but can also improve reliability by leveraging the stability of the existing CDN. CoBlitz demonstrates that this approach can be implemented at low cost, and provides efficient transfers even under heavy load.

Similarly, the Coral module uses a peer-to-peer content distribution network that consists of volunteer sites that run CoralCDN. The CoralCDN sites automatically replicate content as a side effect of users accessing it. A file is retrieved via CoralCDN simply by making a small change to the hostname in an object's URL. Then a peer-to-peer DNS layer transparently redirects browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots. It achieves this through Coral [12], a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a *distributed sloppy hash table* (DSHT).

BitTorrent is a protocol for distributing files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over HTTP is that nodes that download the same file simultaneously also upload portions of the file to each other. This greatly reduces the load on the server and increases scalability. Nodes that upload portions of a file are called *seeds*. BitTorrent employs a *tracker* process to track which portions each seed has and helps clients locate seeds with the portions they need. BitTorrent balances seed loads by having its clients preferentially retrieve unpopular portions, thus creating new seeds for those portions.

Stork also supports traditional protocols such as HTTP and FTP. These protocols contact the repository directly to retrieve the desired data object. It is preferable to use one of the content distribution networks instead of HTTP or FTP as it reduces the repository load.

Stork supports all of these transfer mechanisms with performance data presented in the results section. One key observation is that although these transfer methods are efficient, the uncertainties of the Internet make failure a common case. For this reason the transfer module tries a different transfer mechanism when one fails. For example, if a BitTorrent transfer fails, Stork will attempt CoBlitz, HTTP, or another mechanism until the transfer succeeds or gives up. This

provides efficiency in the common case, and correct handling when there is an error.

Nest Transfer

In addition to the transfer modules listed above, Stork supports a nest transfer module. The nest transfer module provides an additional level of indirection so that the client asks the nest to perform the transfer on its behalf rather than performing the transfer directly. If the nest has a current copy of the requested item in its cache, then it can provide the item directly from the cache. Otherwise, the nest will invoke a transfer module (such as BitTorrent, HTTP, etc.) to retrieve the item, which it will then provide to the client and cache for later use.

Push

Stork supports metadata distribution to the nests using a publish/subscribe system [11]. In a publish/subscribe system, subscribers register their interest in an event and are subsequently notified of events generated by publishers. One such publish/subscribe system is PsEPR [5]. The messaging infrastructure for PsEPR is built on a collection of off-the-shelf instant messaging servers running on PlanetLab. PsEPR publishes events (XML fragments) on channels to which clients subscribe. Behind the scenes PsEPR uses overlay routing to route events among subscribers.

The Stork repository pushes out metadata updates through PsEPR. It also pushes out the repository's metahash file that contains the hashes of the metadata files; this serves as a heartbeat that allows nodes to detect missed updates. In this manner nodes only receive metadata changes as necessary and there is no burden on the repository from unnecessary polling.

Directory Synchronization

In addition to pushing data, Stork also supports a mechanism for pulling the current state from a repository. There are several reasons why this might be necessary, with the most obvious being that the publish/subscribe system is unavailable or has not published data in a timely enough manner. Stork builds upon the transfer modules to create an interface that supports the synchronization of entire directories.

Directory synchronization mirrors a directory hierarchy from the repository to the client. It first downloads the repository's metahash file (the same file that the repository publishes periodically using PsEPR). This file contains a list of all files that comprise the repository's current state and the hashes for those files. Stork compares the hashes to the those of the most recent copies of these files that it has on disk. If a hash does not match, then the file must be re-downloaded using a transfer module.

Share Modules

Virtual machines are a double-edged sword: the isolation they provide can come at the expense of sharing between them. Sharing is used in conventional

systems to provide performance and resource utilization improvements. One example is sharing common application programs and libraries. They are typically installed in a common directory and shared by all users. Only a single copy of each application and library exists on disk and in memory, greatly reducing the demand on these resources. Supporting different versions of the same software is an issue, however. Typically multiple versions cannot be installed in the same common directory without conflicts. Users may have to resort to installing their own private copies, increasing the amount of disk and memory used.

Stork enables sharing in a VM environment by weakening the isolation between VMs to allow file sharing under the control of the nest. Specifically, read-only files can be shared such that individual slices cannot modify the files, although they can be unlinked. This reduces disk and memory consumption. These benefits are gained by all slices that install the same version of a package. It also allows slices to install different package versions in the standard location in their file systems without conflict.

In Stork, sharing is provided via Share modules that hide the details of sharing on different VM platforms. This interface is used by the nest and provides five routines: `init_client`, `authenticate_client`, `share`, `protect`, and `copy`. `init_client` is called when a client binds to the nest, and initializes the per-client state. `Authenticate_client` is used by the nest to authenticate the client that has sent a bind request. This is done by mapping a randomly named file into the client's filesystem and asking it to modify the file in a particular way. Only a legitimate client can modify its local file system, and therefore if the client succeeds in modifying the file the nest requested, the nest knows that it is talking to a legitimate client. The `share` routine shares (or unshares) a file or directory between the client and nest, `protect` protects (or unprotects) a file from modification by the client, and `copy` copies a file between the nest and a client.

The implementation of the Share module depends on the underlying platform. On PlanetLab the Share module communicates with a component of the VMM called Proper [20] to perform its operations. The nest runs in an unprivileged slice – all privileged operations, such as sharing, copying, and protecting files, are done via Proper.

On the Vserver platform the nest runs in the root context, giving it full access to all VM file systems and allowing it to do all of its operations directly. Hard links are used to share files between VMs. The immutable bits are used to protect shared files from modification. Directories are shared using `mount -bind`. Copying is easily done because the root context has access to all VM filesystems.

Package Modules

Stork supports the popular package formats RPM and tar. In the future, other package formats such as

Debian may be added. Each type of package is encapsulated in a package module. Each package module implements the following interfaces:

`is_package_understood`. Returns true if this package module understands the specified package type. Stork uses this function to query each package module until a suitable match is found.

`get_package_provides`. Returns a list of dependencies that are provided by a package. This function is used to generate the metadata that is then used to resolve dependencies when installing packages.

`get_packages_requires`. Returns a list of packages that this package requires. This function is used along with `get_package_provides` to generate the package metadata.

`get_package_files`. Returns a list of the files that are contained in a package. This function is also used when generating package metadata.

`get_package_info`. Returns the name, version, release, and size of a package. This information allows the user to install a specific version of a package.

`get_installed_versions`. Given the name of a package, returns a list of the versions of the package that are installed. This function is used to determine when a package is already installed, so that an installation can be aborted, or an upgrade can be performed if the user has requested upgrades.

`execute_transactions`. Stork uses a transaction-based interface to perform package installation, upgrade, and removal. A transaction list is an ordered list of package actions. Each action consists of a type (install, upgrade, remove) and a package name.

Supported Package Types

`stork_rpm`. Stork currently supports RPM and tar packages. The RPM database is maintained internally by the `rpm` command-line tool, and Stork's RPM package module uses this tool to query the database and to execute the install, update, and remove operations,

`stork_tar`. Tar packages are treated differently because Linux does not maintain a database of installed tar packages, nor is there a provision in tar packages for executing install and uninstall scripts. Stork allows users to bundle four scripts, `.preinstall`, `.postinstall`, `.preremove`, `.postremove` that are executed by Stork at the appropriate times during package installation and removal. Stork does not currently support dependency resolution for tar packages, but this would be a straightforward addition. Stork maintains a database that contains the names and versions of tar packages that are installed that mimics the RPM database provided by the `rpm` tool.

Nest Package Installation

A special package manager, `stork_nest_rpm`, is responsible for performing shared installation of RPM packages. Shared installation of tar packages is not supported at this time. Performing a share operation is a three-phase process.

In the first phase, `stork_nest_rpm` calls `stork_rpm` to perform a private installation of the package. This allows the package to be installed atomically using the protections provided by RPM, including executing any install scripts. In the second phase, `stork_nest_rpm` contacts the Stork nest and asks it to prepare the package for sharing. The prepare module is discussed in the following section. Finally, in the third phase `stork_nest_rpm` contacts the nest and instructs it to share the prepared package. The nest uses the applicable share module to perform the sharing. The private versions of files that were installed by `stork_rpm` are replaced by shared versions. Stork does not attempt to share configuration files because these files are often changed by the client installation. Stork also examines files to make sure they are identical prior to replacing a private copy with a shared copy.

Removal of packages that were installed using `stork_nest_rpm` requires no special processing. `stork_nest_rpm` merely submits the appropriate remove actions to `stork_rpm`. The `stork_rpm` module uses the `rpm` tool to uninstall the package, which unlinks the package's files. The link count of the shared files is decremented, but is still nonzero. The shared files persist on the nest and in any other clients that are linked to them.

Prepare Modules

Prepare modules are used by the nest to prepare a package for sharing. In order to share a package, the nest must extract the files in the package. This extraction differs from package installation in that no installation scripts are run, no databases are updated, and the files are not moved to their proper locations. Instead, files are extracted to a sharing directory.

Prepare modules only implement one interface, the `prepare` function. This function takes the name of a package and the destination directory in which to extract the package.

RPM is the only package format that Stork currently shares. The first step of the `stork_rpm_prepare` module is to see if the package has already been prepared. If it has, then nothing needs to be done. If the package has not been prepared, then `stork_rpm_prepare` uses `rpm2cpio` to convert the RPM package into a `cpio` archive that is then extracted. `stork_rpm_prepare` queries the `rpm` tool to determine which files are configuration files and moves the configuration files to a special location so they will not be shared. Finally, `stork_rpm_prepare` sets the appropriate permissions on the files that it has extracted.

Stork Walkthrough

This section illustrates how the Stork components work together to manage packages using the earlier example in which an administrator installs an updated version of the `foobar` package on the VMs the company uses for testing and on the non-VM desktop machines used by the company's developers.

1. The administrator uses `storkutil` to add the new version of the `foobar` package to her TP file.
2. She uses `storkutil` to add the groups `Devel` and `Test` to her `groups.pacman` file, representing the developer's end systems and the testing VMs, respectively. Since groups can be reused, this step most likely would have been done previously.
3. The administrator uses `storkutil` to add a line to her `packages.pacman` file instructing the `Test` group to update `foobar`. She does the same for the `Devel` group.
4. `Storkutil` automatically signed these files with her private key. She now uploads these files to a Stork repository. If the new version of the `foobar` package is not already on the repository she uploads this as well.
5. The repository treats the TP and `pacman` files similarly. The signatures are verified using the administrator's public key that is embedded in the file name. The new files replace the old if their signatures are valid and their timestamps newer. The `foobar` package is stored in a directory whose name is its secure hash. The package metadata is extracted and made available for download.
6. The repository uses the publish/subscribe system `PsEPR` to push out a new repository meta-hash to the VMs.
7. The VMs are running `stork_receive_update` and obtain the new repository meta-hash. The `stork_receive_update` daemon wakes up the `pacman` daemon.
8. The `pacman` daemon updates its metadata. On non-VM platforms, the files are downloaded efficiently using whatever transfer method is listed in the Stork configuration file. On VM platforms, `pacman` retrieves the files through the nest (which means the files are downloaded only once per physical machine).
9. `Pacman` processes its metadata and if the current VM is in either the `Test` or `Devel` groups it calls `stork` to update the `foobar` package.
10. The `stork` tool verifies that it has the current metadata and configuration files. This is useful because it is not uncommon for several files to be uploaded in short succession. If this is not the case it retrieves the updated files in the same manner as `pacman`.
11. Stork verifies that the specified version of `foobar` is not already installed; if it is, Stork simply exits.
12. Stork searches the package metadata for the specified package. If no candidate is found then it exits with an error message that the package cannot be found. Multiple candidates may be returned if the metadata database contains several versions of `foobar`.
13. Stork verifies that the user trusts the candidate versions of `foobar`. It does this by applying the

rules from the user's TP file one at a time until a rule is found that matches each candidate. If the rule is a DENY rule, then the candidate is rejected. If the rule is an ACCEPT rule, then the candidate is deemed trustworthy. The result of trust verification is an ordered list of package candidates.

14. Stork now has one or more possible candidates for foobar. However, if foobar depends on other packages stork repeats steps 13-17 for the dependencies to determine if those dependencies can be satisfied.
15. Stork now has a list of packages that are to be updated, including foobar and its missing dependencies. Stork uses a transfer module to retrieve foobar and dependent packages. The highest priority transfer method is to contact the repository, which is via the nest in VM environments.
16. In a VM environment the nest receives the requests for foobar and its dependencies from the client VM. If these files are already cached on the nest, then the nest provides those local copies. If not, then the nest invokes the transfer modules (BitTorrent, CoBlitz, etc.) to retrieve the files. When retrieval is complete, the nest shares the package with the client VM.
17. Stork now has local copies of foobar and its dependent packages. The client queries the package modules to find one that can install the package. In non-VM environments the `stork_rpm` module installs the packages using RPM and returns to stork which exits. In VM environments the `stork_nest_rpm` module is tried first (stork will fail over and use `stork_rpm` if this module fails). Because foobar is an RPM package, `stork_nest_rpm` can process it. Stork builds a transaction list and passes it to the `execute_transactions` function of `stork_nest_rpm`.
18. In a VM environment the `stork_nest_rpm` module passes the transaction list to `stork_rpm` in order to install a private non-shared copy of the foobar package.
19. In a VM environment the `stork_nest_rpm` module then contacts the nest and issues a request to prepare and share foobar. The nest uses the appropriate prepare module to extract the files contained in foobar. The nest uses the appropriate share module to share the extracted files with the client VM. Sharing overwrites the private versions of the files in the client's VM with shared versions from the foobar package.

In some cases there will be systems that do not receive the PsEPR update. This could occur because PsEPR failed to deliver the message or perhaps because the system is down. If PsEPR failed then `pacman` check for updates every five minutes. If the system was down then when it restarts `pacman` will run. Either way `pacman` will start and obtain a new repository

metahash and the system will continue the process from Step 8.

If nest or module failures happen, stork fails over to other modules that might be able to service the request. For example, if the packages cannot be downloaded by BitTorrent, the tool will instead try another transfer method like CoBlitz as specified in the master configuration file.

Results

Stork was evaluated via several experiments on PlanetLab. The first measures the effectiveness of Stork in conserving disk space when installing packages in VM environments. The second experiment measures the memory savings Stork provides to packages installed in multiple VMs. The final set of experiments measure the impact Stork has on package downloads both in performance and in repository load.

Disk Usage

The first experiment measured the amount of disk space saved by installing packages using Stork versus installing them in client slices individually (Figure 5). These measurements were collected using the 10 most popular packages on a sample of 11 PlanetLab nodes. Some applications consist of two packages: one containing the application and one containing a library used exclusively by the application. For the purpose of this experiment they are treated as a single package.

Rank	Package Name	Disk Space (KB)		Percent Savings
		Standard	Stork	
1	scriptroute	8644	600	93%
2	undns	13240	652	95%
3	chord	64972	1216	98%
4	j2re	61876	34280	45%
5	stork	320	32	90%
6	bind	6884	200	97%
7	file	1288	36	97%
8	make	808	32	96%
9	c++	3220	44	99%
10	binutils	6892	60	99%

Figure 5: Disk Used by Popular Packages. This table shows the disk space required to install the 10 most popular packages installed by the slices on a sampling of PlanetLab nodes. The *Standard* column shows how much per-slice space the package consumes if nothing is shared. The *Stork* column shows how much per-slice space the package requires when installed by Stork.

For all but one package, Stork reduced the per-client disk space required to install a package by over 90%. It should be noted that the nest stores an entire

copy of the package to which the clients link; Stork's total space savings is therefore a function of the total number of clients sharing a package.

One package, `j2re`, had savings of only 45%. This was because many of the files within the package were themselves inside of archives. The post-install scripts extract these files from the archives. Since the post-install scripts are run by the client, the nest cannot share the extracted files between slices. By repackaging the files so that the extracted files are part of the package, this issue can be avoided.

Memory Usage

Stork also allows processes running in different slices to share memory because they share the underlying executables and libraries (Figure 6). The primary application was run from each package and its memory usage was analyzed. It was not possible to get memory sharing numbers directly from the Linux kernel running on the PlanetLab nodes. Since the PlanetLab kernel shares free memory pages between VMs and there are many VMs being used by different users on each PlanetLab node, this increases the difficulty of gathering accurate memory usage information.

To obtain approximate results the `pmap` command was used to dump the processes' address spaces. Using the page map data, it is possible to classify memory regions as shared or private. The results are only approximate, however, because the amount of address space shared does not directly correspond to the amount of memory shared as some pages in the address space may not be resident in memory. More accurate measurements require changes to the Linux kernel that are not currently feasible.

Another difficulty in measuring memory use is that it changes as the program runs. Daemon programs

were simply started and measured. Applications that process input files (such as `java` and `make`) were started with a minimal file that goes into an infinite loop. The remaining applications printed their usage information and were measured before they exited.

The resulting measurements show that Stork typically reduces the memory required by additional processes by 50% to 60%. There are two notable exceptions: `named` and `java`. These programs allocate huge data areas that are much larger than their text segments and libraries. Data segments are private, so this shadows any benefits Stork provides in sharing text and libraries.

Package Retrieval

Stork downloads packages to the nest efficiently, in terms of the amount of network bandwidth required, server load, and elapsed time. This was measured by retrieving a 10 MB package simultaneously from 300 nodes (Figure 7), simulating what happens when a new package is stored on the repository. Obviously faulty nodes were not included in the experiments, and a new randomly-generated 10 MB file was used for each test. Each test was run three times and the results averaged. It proved impossible to get all 300 nodes to complete the tests successfully; in some cases some nodes never even started the test. Faulty and unresponsive nodes are not unusual on PlanetLab. This is dealt with by simply reporting the number of nodes that started and completed each test.

Repository load is important to system scalability, represented as the total amount of network traffic generated by the repository. This includes retransmissions, protocol headers, and any other data. For BitTorrent, this includes the traffic for both the tracker and the initial seed as they were run on the same node;

Rank	Package Name	Application Name	Memory (MB)		Percent Savings
			Standard	Stork	
1	scriptroute	srinterpreter	5.8	3.2	45%
2	undns	undns_decode	4.2	2.0	53%
3	chord	adbd	7.6	2.3	70%
3	chord	lsd	7.5	1.1	86%
4	j2re	java	206.8	169.5	18%
5	stork	stork	3.4	1.2	64%
6	bind	named	36.7	32.1	12%
7	file	file	2.6	1.3	50%
8	make	make	2.5	1.1	54%
9	cpp	cpp	2.5	1.2	52%
10	binutils	objdump	3.3	1.4	59%
10	binutils	strip	2.9	1.0	65%
10	binutils	strings	3.4	1.7	50%

Figure 6: Memory Used by Popular Packages. Packages installed by Stork allow slices to share process memory. The *Standard* column shows how much memory is consumed by each process when nothing is shared. With Stork the first process will consume the same amount as the *Standard* column, but additional processes only require the amount shown in the *Stork* column.

running them on different nodes made negligible difference. At a minimum the repository must send 10 MB, since the clients are downloading a 10 MB file. CoBlitz generated the least network traffic, sending 7.8 times the minimum. BitTorrent sent 3.3 times as much data as CoBlitz and Coral sent 5.5 times as much as CoBlitz. HTTP was by far the worst, sending 39.5 times more than CoBlitz. In fact, HTTP exceeded the product of the number of clients and the file size because of protocol headers and retransmissions.

For each test the amount of useful bandwidth each client received (file data exclusive of network protocol headers) is reported, including both the median and mean, as well as the 25th and 75th percentiles. BitTorrent's mean bandwidth is 2.8 times that of CoBlitz, 3.3 times that of HTTP, and 4.2 times that of Coral. HTTP does surprisingly well, which is a result of a relatively high-speed connection from the repository to the PlanetLab nodes.

Figure 8 shows the cumulative distribution of client completion times. More than 78% of the nodes completed the transfer within 90 seconds using BitTorrent, compared to only 40% of the CoBlitz and 23% of the Coral nodes. None of the HTTP nodes finished within 90 seconds.

The distribution of client completion times also varied greatly among the protocols. The time of HTTP varied little between the nodes: there is only an 18% difference between the completion time of the 25th and 75th percentiles. The BitTorrent clients in the 25th percentile finished in 48% the time of clients in the 75th percentile, while Coral clients differed by 64%. CoBlitz had the highest variance, so that the clients in the 25th percentile finished in 14% of the time of the clients in the 75th percentile, meaning that the slowest nodes took 7.3 times as long to download the file as the fastest.

These results reflect how the different protocols download the file. All the nodes begin retrieving the file at the same time. Clients in BitTorrent favor downloading rare portions of the file first, which leads to most of the nodes downloading from each other, rather than from the repository. The CoBlitz and Coral CDN nodes download pieces of the file sequentially. This causes the clients to progress lock-step through the file, all waiting for the CDN node with the next piece of the

file. This places the current CDN node under a heavy load while the other CDN nodes are idle.

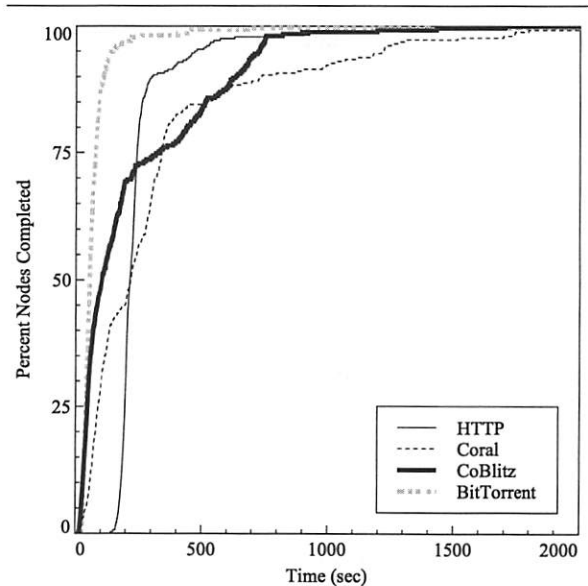


Figure 8: Elapsed Time. This graph shows the cumulative distribution of client completion times. Only nodes that successfully completed are included.

Based on these results Stork uses BitTorrent as its first choice when performing package retrievals, switching to other protocols if it fails. BitTorrent decreased the transfer time by 70% over HTTP and reduces the amount of data that the repository needs to send by 92%.

Related Work

Prior work to address the problem of software management can be roughly classified into three categories: (a) traditional package management systems which resolve package dependencies and retrieve packages from remote systems, (b) techniques to reduce the cost of duplicate installs, and (c) distributed file systems that are used for software distribution.

Traditional Package Management Systems

Popular package management systems [2, 10, 27, 34, 36] typically retrieve packages via HTTP or FTP,

Transfer Protocol	Effective Client Bandwidth (Kbps)				Nodes Completed	Server MB Sent
	25%	Median	Mean	75%		
HTTP	413.9	380.6	321.2	338.1	280/286	3080.3
Coral	651.3	468.9	253.6	234.1	259/281	424.7
CoBlitz	1703.5	737.2	381.1	234.0	255/292	77.9
BitTorrent	2011.8	1482.2	1066.9	1044.0	270/284	255.8

Figure 7: Package Download Performance. This table shows the results of downloading a 10 MB file to 300 nodes. Each result is the average of three tests. The client bandwidth is measured with respect to the amount of file data received, and the mean, median, 25th percentile, and 75th percentile results given. The *Nodes Completed* column shows the number of nodes that started and finished the transfer. The *Server MB Sent* is the amount of network traffic sent to the clients, including protocol headers and retransmissions.

resolve dependencies, and manage packages on the local system. They do not manage packages across multiple machines. This leads to inefficiencies in a distributed VM environment because a service spans multiple physical machines, and each physical machine has multiple VMs. The package management system must span nodes and VMs, otherwise VMs will individually download and install packages, consuming excessive network bandwidth and disk space.

Most package management systems have support for security. In general, however, the repository is trusted to contain valid packages. RPM and Debian packages can be signed by the developer and the signature is verified before the package is installed. This requires the user to have the keys of all developers. In many cases package signatures are not checked by default because of this difficulty. The trustedpackages file mechanism in Stork effectively allows multiple signatures per package so that users require fewer keys.

Reducing the Cost of Duplication

Most VMMs focus on providing isolation between VMs, not sharing. However different techniques have been devised to mitigate the disk, memory, and network costs installing duplicate packages.

Disk A good deal of research has gone into preventing duplicate data from consuming additional disk space. For example, many file systems use copy-on-write techniques [6, 8, 14, 15, 16, 30] which allow data to be shared but copied if modified. This allows different “snapshots” of a file system to be taken where the unchanged areas will be shared amongst the “snapshots”. However, this does not combine identical files that were written at different locations (as would happen with multiple VMs downloading the same package).

Some filesystem tools [4] and VMMs [17, 18] share files that have already been created on a system. They unify common files or blocks to reduce the disk space required. This unification happens after the package has been installed; each VM must download and install the package, only to have its copies of the files subsequently replaced with links. Stork avoids this overhead and complexity by linking the files in the first place.

Another technique for reducing the amount of storage space consumed by identical components detects duplicate files and combines them as they are written [25]. This is typically done by using a hash of the file blocks to quickly detect duplicates. Stork avoids the overhead of needing to check file blocks for duplicates on insertion and avoids the need to download the block multiple times in the first place.

Memory There are many proposals that try to reduce the memory overhead of duplicate memory pages. Disco [6] implements copy-on-write memory sharing between VMs which allows not only a process’ memory pages to be shared but also allows duplicate buffer cache pages to be shared. The sharing

provided by Stork is much less effective than Disco, but at a much lower cost.

Stork allows VMs to share the memory used by shared applications and libraries. VMware ESX Server [32] also allows VMs to share memory, but does so based on page content. A background process scans memory looking for multiple copies of the same page. Any redundant copies are eliminated by replacing them with a single copy-on-write page. This allows for more potential sharing than Stork, as any identical pages can be shared, but at the cost of having processes create duplicate pages only to have them culled.

Network Bandwidth A common technique to mitigate the network costs of duplicate data retrieval is to use a proxy server [7, 26, 28, 35]. Proxy servers minimize the load on the server providing the data and also increase the performance of the clients. However, the data still must be transferred multiple times over the network, while the Stork nest provides the data to the client VMs without incurring network traffic (much like each system running its own proxy server for packages). Stork uses techniques such as P2P file dissemination [9] along with proxy based content retrieval [22, 12] to minimize repository load.

Distributed File Systems

Stork uses content distribution mechanisms to download packages to nodes. Alternatively, a distributed file system such as NFS could be used. For example, the relevant software package files could be copied onto a file system that is shared via NFS. There are many drawbacks to this technique including poor performance and the difficulty in supporting different (and existing) packages on separate machines.

Among the numerous distributed files systems Shark [1] and SFS-RO [13] are two that have been promoted as a way to distribute software. Clients can either mount applications and libraries directly, or use the file system to access packages that are installed locally. The former has performance, reliability, and conflict issues; the latter only uses the distributed file system to download packages, which may not be superior to using an efficient content distribution mechanism and does not provide centralized control and management.

Conclusion

Stork provides both efficient inter-VM package sharing and centralized inter-machine package management. When sharing packages between VMs it typically provides over an order of magnitude in disk savings, and about 50% of the memory costs. Additionally, each node needs only download a package once no matter how many VMs install it. This reduces the package transfer time by 70% and reduces the repository load by 92%.

Stork allows groups of VMs to be centrally administered. The pacman tool and its configuration

files allow administrators to define groups of VMs and specify which packages are to be installed on which groups. Changes are pushed to the VMs in a timely fashion, and packages are downloaded to the VMs efficiently. Stork has been in use on PlanetLab for over four years and has managed thousands of virtual machines. The source code for Stork may be downloaded from <http://www.cs.arizona.edu/stork>

Acknowledgments

First and foremost, we would like to thank all of the undergraduates who were not coauthors but assisted with the development of Stork including Mario Gonzalez, Thomas Harris, Seth Hollyman, Petr Moravsky, Peter Peterson, Justin Samuel, and Byung Suk Yang. We would also like to thank all of the Stork users. A special thanks goes out to the developers of the services we use including Vivek Pai, KyoungSoo Park, Sean Rhea, Ryan Huebsch, and Robert Adams for their efforts in answering our countless questions. We would especially like to thank Steve Muir at PlanetLab Central for his efforts on our behalf throughout the development of Stork and Proper.

Biographies

Justin Cappos is a Ph. D. student in the Computer Science Department at the University of Arizona. He has been working on projects involving large, real world distributed systems for the past four years. His other research interests include resource allocation, content aggregation, and tools for building distributed systems. He can be reached electronically at justin@cs.arizona.edu.

Scott Baker received a B.S., M.S., and Ph.D. in Computer Science at the University of Arizona. He now works as a software consultant, with a focus in Linux systems programming. He can be reached at bakers@cs.arizona.edu.

Jeremy Plichta is a senior at the University of Arizona majoring in Computer Science, with a minor in Mathematics. After graduating, he plans to pursue a career in industry with the possibility of graduate study at a later date. He designed and maintained the Stork Repository as well as some aspects of the Stork GUI. He can be reached electronically at jplichta@arizona.edu.

Duy Nguyen is currently an undergraduate student at the University of Arizona. He has been working on the Stork Project for a year. His other interests include programming languages, networking, web design, animation, and instructional applications. He can be reached electronically at dqn@email.arizona.edu.

Jason Hardies received a BA in linguistics at the University of Arizona. While a student he worked on the Stork project. After leaving the university in 2006, he joined the healthcare software company Epic Systems, Corp. in Madison, WI where he is a software

developer. He can be reached electronically at jhardies@epicsystems.com.

Matt Borgard is currently an undergraduate at the University of Arizona, studying Computer Science and Creative Writing. His interests include storage, computational linguistics and interactive storytelling. He can be reached electronically at mborgard@email.arizona.edu.

Jeffrey Johnston received a B.S. in Computer Science at the University of Arizona in 2007. He is currently employed at IBM in Tucson, Arizona where he is a software engineer in the z/OS Storage DFSMSHsm department. He can be reached electronically at stork@kidsquid.com.

John H. Hartman is an Associate Professor in the Department of Computer Science at the University of Arizona, which he joined in 1995. He received his Ph.D. in Computer Science from the University of California at Berkeley in 1994. His research interests include distributed file systems, networks, distributed operating systems, and mobile computing. He can be reached electronically at jhh@cs.arizona.edu.

Bibliography

- [1] Annapureddy, S., M. J. Freedman, and D. Mazières, "Shark: Scaling File Servers via Cooperative Caching," *Proceedings 2nd NSDI* Boston, MA, May, 2005.
- [2] *Debian APT tool ported to RedHat Linux*, <http://www.apt-get.org/>.
- [3] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proceedings 19th SOSP*, Lake George, NY, Oct, 2003.
- [4] Bolosky, W. J., S. Corbin, D. Goebel, and J. R. Douceur, "Single Instance Storage in Windows 2000," *Proceedings 4th USENIX Windows Systems Symposium*, pp. 13-24, Seattle, WA, Aug, 2000.
- [5] Brett, P., R. Knauerhase, M. Bowman, R. Adams, A. Nataraj, J. Sedayao, and M. Spindel, "A Shared Global Event Propagation System to Enable Next Generation Distributed Services," *Proceedings of the 1st Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec, 2004.
- [6] Bugnion, E., S. Devine, K. Govil, and M. Rosenblum, "Disco: Running Commodity Operating Systems On Scalable Multiprocessors," *ACM Transactions on Computer Systems*, Vol. 15, Num. 4, pp. 412-447, Nov, 1997.
- [7] Chankhunthod, A., P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A Hierarchical Internet Object Cache," *USENIX Annual Technical Conference*, pp. 153-164, 1996.
- [8] Chutani, S., O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham,

- "The Episode File System," *Proceedings of the USENIX Winter 1992 Technical Conference*, pp. 43-60, San Francisco, CA, USA, 1992.
- [9] Cohen, B., "Incentives Build Robustness in BitTorrent," *Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [10] Debian - dpkg, <http://packages.debian.org/stable/base/dpkg>.
- [11] Eugster, P. T., P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, Vol. 35, Num. 2, pp. 114-131, Jun, 2003.
- [12] Freedman, M. J., E. Freudenthal, and D. Mazières, "Democratizing Content Publication with Coral," *Proceedings 1st NSDI*, San Francisco, CA, Mar., 2004.
- [13] Fu, K., M. F. Kaashoek, and D. Mazières, "The Click Modular Router," *ACM Transactions on Computer Systems*, Vol. 20, Num. 1, pp. 1-24, Feb, 2002.
- [14] Ghemawat, S., H. Gobioff, and S.-T. Leung, "The Google File System," *Proceedings 19th SOSP*, Lake George, NY, Oct 2003.
- [15] Hitz, D., J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 235-246, San Francisco, CA, USA, 1994.
- [16] Howard, J. H., M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, N., and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computing Systems*, Vol. 6, Num. 1, pp. 51-81, 1988.
- [17] Kamp, P.-H., and R. N. M. Watson, "Jails: Confining the Omnipotent Root," *Proceedings 2nd International SANE Conference*, Maastricht, The Netherlands, May, 2000.
- [18] Linux VServers Project, <http://linux-vservers.org/>.
- [19] Mazières, D., M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating Key Management From File System Security," *Proceedings 17th SOSP*, pp. 124-139, Kiawah Island Resort, SC, Dec, 1999.
- [20] Muir, S., L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman, "Proper: Privileged Operations in a Virtualised System Environment," *Proceedings USENIX '05*, Anaheim, CA, Apr, 2005.
- [21] Park, K., and V. S. Pai, "Deploying Large File Transfer on an HTTP Content Distribution Network," *Proceedings of the 1st Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec, 2004.
- [22] Park, K., and V. S. Pai, "Scale and Performance in the CoBlitz Large-File Distribution Service," *Proceedings 3rd NSDI*, San Jose, CA, May, 2005.
- [23] Peterson, L., T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," *Proceedings Hot-Nets-I*, Princeton, NJ, Oct, 2002.
- [24] PlanetLab, <http://www.planet-lab.org>.
- [25] Quinlan, S., and S. Dorward, "Venti: A New Approach to Archival Storage," *First USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002.
- [26] Rabinovich, M., J. Chase, and S. Gadde, "Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network," *Computer Networking ISDN Systems*, Vol. 30, pp. 2253-2259, 1998.
- [27] RPM Package Manager, <http://www.rpm.org/>.
- [28] squid: Optimising Web Delivery, <http://www.squid-cache.org/>.
- [29] Stork Project, <http://www.cs.arizona.edu/stork/>.
- [30] Thekkath, C. A., T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pp. 224-237, ACM Press, New York, NY, USA, 1997.
- [31] VMWare Workstation, <http://www.vmware.com/>.
- [32] Waldspurger, C. A., "Memory Resource Management in VMware ESX Server," *Operating Systems Review*, Vol. 36, pp. 181-194, 2002.
- [33] Wang, L., K. Park, R. Pang, V. Pai., and L. Peterson, "Reliability and Security in the CoDeeN Content Distribution Network," *Proceedings USENIX '02*, San Francisco, CA, Aug, 2002.
- [34] Windows Update, <http://update.windows.com/>.
- [35] WinProxy, <http://www.winproxy.com/index.asp>.
- [36] Yum: Yellow Dog Updater Modified, <http://linux.duke.edu/projects/yum/>.

Decision Support for Virtual Machine Re-Provisioning in Production Environments

Kyrre Begnum and Matthew Disney – Oslo University College, Norway

Æleen Frisch – Exponential Consulting

Ingard Mevåg – Oslo University College

ABSTRACT

Management of virtual machines (VMs) on a large scale remains a significant challenge today. We lack general and vendor-independent quantitative criteria/metrics by which to describe the state of infrastructures for virtualization. This data is essential to both expressing administrative policy goals and to measure ongoing compliance with them in production settings.

In this work, we consider VM management in production environments. We investigate VM applicability to real-world scientific computing problems by comparing application performance in a controlled environment of physical servers with the Manage Large Networks (MLN) tool's implementation of the same scenario on virtual machines. Based on our observations, we propose three new metrics by which to describe and analyze the infrastructure in order to incorporate virtual machine management closer into policy. The metrics have been implemented into the latest release of our MLN tool for virtual machine management.

Introduction

Most approaches to virtualization devote the bulk of the efforts on their features related to deployment and fail-over mechanisms. Indeed, much of the work in this area focuses on these areas exclusively. However, after running a virtual infrastructure in production, one discovers how deployment decisions and virtual machine behavior affect the overall performance in ways which are not completely described nor obviously transparent. One key challenge is avoiding resource conflicts among the virtual machines. This task depends on many local factors, such as the number and behavior of the virtual machines, as well as the number and capacity of physical servers and other infrastructure resources like common storage. We use the term “virtual(ized) infrastructure” as a general scenario where more than one physical servers are used to host a range of virtual machines with varying life-span and purposes. We do not intend to refer to a particular virtual machine management framework or product.

The system administrator needs methods to analyze and describe the site's virtualization infrastructure in a technology independent way. Such analysis would assist in the following tasks:

- Determine the level of redundancy in server capacity for downtime planning.
- Review the level of resource conflicts between virtual machines in order to identify and remove bottlenecks.
- Find the optimal server location for new virtual machines in the infrastructure.
- Identify which virtual machines are apt to demand resources at the same time in order to separate them from each other.

Possessing this and related data will also enable the system administrator to express and implement explicit, quantitative policy rules for the site, enabling her to address a variety of concerns: How should one describe the desired level of redundancy in server capacity and measure its compliance? Could a level of resource conflicts function as an indication of how well the virtual machines are deployed across the site? This is not only valuable for system administrators today, but necessary steps towards autonomic capabilities of future management tools, since self-optimizing system behavior techniques require quantifiable metrics in order to measure success.

MLN (Manage Large Networks) [1] is a virtual machine management tool developed at the University College of Oslo. It supports both Xen and User-Mode Linux. After using MLN in various scenarios and addressing the need to efficiently deploy large scenarios of virtual machines [2, 3, 4], we have arrived at the point where long-time management reveals new challenges, problems which are still unexplored by the community. This article addresses these challenges by proposing three methods by which to analyze a virtualized infrastructure. The *Server redundancy level*, *Resource conflict matrix* and *Location conflict table* are technology independent measures of the state of the infrastructure.

This text is organized as follows: We first demonstrate the viability of using Xen virtual machines and MLN in a production environment. This case study demonstrated to us the challenges of long-time management of virtual machines. Next, our approaches for analysis are presented and discussed in detail. We then discuss the implementation of these in MLN and

discuss our findings to date. Finally, we outline future work and review related work.

VM Viability in Production Scientific Computing Environments

In recent years, there have been many calls for the increased use of virtualization technologies for high performance computing (HPC) and especially scientific modeling and simulations (see, e.g., [5, 6]). The many advantages of virtualization and virtual machines (VMs) usually apply to such specialized computing environments as they do to general purpose systems (e.g., the ability to control resource usage, security barriers, performance and reliability improvements due to isolation of distinct processes, and so on).

The computing environment and typical job characteristics of scientific computing environments, whether located in academic institutions, corporate research and development or government laboratories, share many common traits which distinguish them from more general computing environments:

- Rather than a mix of many jobs of various types and needs, scientific computing typically consists of a few computations of significant duration. The most intensive of scientific computing applications have essentially unbounded need for CPU cycles, physical memory and memory bandwidth, and, in some cases, disk and/or network I/O capacity.
- Research efforts tend to rely on a few software packages related to the field under investigation. Most production software is commercial for which source code is not available (and the expertise required to modify it is not present at most sites).
- Research groups prefer to have their own computing resources, with typically little or no system administrative support from any centralized IT organization.

Such environments face many challenges which virtualization technology, in conjunction with a tool like MLN, can address:

- Deploying idle computing resources, where and when they are available. This can include applying general purpose computers to simulation problems after hours. VMs allow the host operating system environment to remain unaffected by allowing scientific computing usage of the hardware.
- Since scientific computations can execute for hours, days or even weeks, being able to start, pause, restart and migrate such jobs is very beneficial. VMs and MLN provide this capability to any application which employs their resources, often adding this valuable feature for the first time. For example, Gaussian 03, the production code we employed in performance tests, has only limited job restarting capabilities. MLN

allows jobs to be paused at any point. In the past, such capabilities were present on in special purpose checkpointing libraries [7, 8].

- Multiple operating system environments – even legacy ones – can be maintained and invoked on the same hardware. This is important in this arena in that distinct software packages typically have limited, and often contradictory, operating system version support (often lagging well behind the current releases). MLN allows distinct OS environments to be set up and cloned easily, ready to be reused as often and for as long as needed.

Despite these very real benefits, however, performance is still the most important consideration in scientific computing, so virtualization will need to come with few associated resource costs in order to be adopted. Previous work has focused on the performance achieved on standard benchmarks [9, 10, 11, 12]. While this data is interesting, and in general encouraging with respect to VM use for HPC, the limitations of benchmarks in modeling actual scientific computing are well known. First, such benchmarks tend to focus on single performance metrics in isolation: CPU speed, memory bandwidth, disk I/O, network message passing, and so on. Secondly, the algorithms employed in the computational portions are among the simplest of those used in real applications (e.g., Linpack). In addition, the results of the more complex, higher level benchmarks (e.g., the SPEC suite, the GCM program used in [10], and the like), the results are reduced to a single metric, megaflops achieved, which has been repeatedly shown to bear only a vaguely proportionate relationship to actual production performance. Finally, and often most importantly, the problem size is far too small to be useful or representative of actual computations and computing requirements.

Accordingly, we chose to conduct some tests with actual production code on modest sized but realistic problems. We used the Gaussian 03 computational chemistry package. Gaussian 03 performs electronic structure calculations, modeling the properties of chemical compounds and reactions. It is widely used by academic and industrial chemists, chemical engineers, biochemists, physicists, and materials scientists throughout the world, addressing one of the key Grand Challenge level problems. Computationally, it is a very demanding and rigorous application whose achieved performance depends on the combination of CPU performance, memory bandwidth and, for some simulations, disk I/O transfer rates.

We ran three jobs of increasing CPU requirements in several ways: under a standard Red Hat Enterprise Linux kernel, directly under a Xen version of that operating system, and in a Xen virtual machine (details, which are probably understandable only to chemists, are given in a separate section below). We

also ran the jobs on a single node and in a parallel mode using three nodes; the parallel computations were performed under the Xen-enabled operating system and in VMs. In this way, we could test the performance effects of both aspects of virtualization: the modified Linux kernel and using a VM itself. The three nodes chosen for the parallel computations were quite different in performance characteristics and available resources. This selection was made to model the performance that might be obtained from drawing together idle systems in an ad hoc manner after hours. However, parallel performance is known to be best when using symmetric nodes; this approach thus represents a worst-case scenario in that the performance of the more powerful systems is reduced to that of the weakest node.

The following table gives the parallel speedups obtained in the two environments (comparisons are with respect to a single node of the same type):

Parallel Speedups (1 vs. 3 Nodes)		
Job	Xen	VM
1	2.1	2.1
2	2.8	2.8
3	1.2	1.3

Table 1: Parallel speedup comparison.

Job 1, the shortest job, obtains reasonable parallel speedups, and Job 2 does quite well (as the maximum speedup is 3.0); both of these jobs have few I/O requirements. This is in contrast to Job 3, chosen because of its substantial I/O requirements. Even in this case, however, parallelization provides some performance benefits. For our purposes, however, the key result in the preceding table is that using virtual machines for the computations produced identical performance to the jobs run directly on the hardware. Using a VM had no adverse affect on parallelization efficiency.

The following table explores the overhead associated with virtualization – and specifically the Xen approach – in more detail:

Virtualization Overhead		
Job	Xen over RHEL	VM over RHEL
1	3.8%	0.6%
2	3.9%	0.8%
3	3.1%	7.9%

Table 2: Virtual overhead comparison.

The columns compare the performance for single processor jobs in the three environments: directly on the hardware with a standard kernel (RHEL), directly on the hardware with a Xen-enabled kernel (Xen) and in a Xen VM. The table indicates that when disk I/O is not a factor, then there is only a quite small performance penalty associated with the Xen operating system and virtually none associated with using a VM,

both compared to the vanilla Linux OS. Interestingly, the Xen kernel itself slightly less efficient than the vanilla operating system running in a VM.

When disk I/O is a significant factor, the results are somewhat different. The overhead associated with the Xen-enabled OS remains more-or-less constant, but there is addition associated with running in a VM (at least 4.8% for this job). The total overhead of about 8% is still quite acceptable, but reducing that level would be desirable. Our initial tests focused on making virtualization configuration and use as easy as possible. Thus, the VM was built on a logical volume. In the jobs running directly under the Xen-enabled OS, however, disk I/O associated with the computation was to a logical volume as well, and the LVM undoubtedly imposed some overhead.

Future work will look at minimizing I/O overhead in the VM environment while still retaining VM configuration and build simplicity. However, these results show clearly that performance in an MLN-managed virtual environment is quite acceptable for production Gaussian 03 calculations.

New Challenges and Approaches

Our infrastructure consisted of a total of 11 servers spread over two separate locations. Although the performance levels of Xen were acceptable, we encountered management issues which revealed new challenges:

- Previous work has shown that there is a substantial performance degradation if two virtual machines compete for the same resources on the same server [3]. We encountered difficulties avoiding such conflicts due to lack of overview over the entire infrastructure.
- At one point, one of the servers became unstable and crashed. This taught us a valuable lesson because we ended up with insufficient capacity to host all of our virtual machines until the problem was fixed. We believe that a way to plan ahead for these eventualities would avoid the same situation in the future.
- There was no decision support from MLN to find the optimal placement for a new virtual machine project. We had to manually inspect the other running projects to arrive at a solution.

We believe that these challenges are universal for all system administrators who have to manage sufficiently many servers and virtual machines. Let us first describe a virtualized infrastructure in a more generalized way.

The foundation of a virtualized infrastructure is its physical servers, for which we will use the term *server*, and the virtual machines (*VMs*). We make two basic assumptions:

1. The infrastructure consists of more than one physical server.

2. The infrastructure enables the system administrator to move virtual machines between the physical servers.

What is left is to find the optimal placement of the virtual machines such that the load is evenly distributed and that all the VMs experience a satisfactory level of performance. This process is highly dependent on the organizations context. The resulting *provisioning policy* will act as a local guidebook for decision making or a policy for which an expert system may surveil and tune the infrastructure.

It is essentially the system administrator's duty to re-provision virtual machines for the best performance. This manual process depends on information about the infrastructure and the individual virtual machines. In some cases, knowledge about the virtual machine can help the decision process. Is it planned to be a web-service or a shell server? Can we expect performance peaks at specific times? Over what time period will the virtual machine run? This information may be hidden from the administrator if users can create their own virtual machines independently. What remains is the static hardware description of the virtual machines resource consumption and its performance profile. We will take this perspective in this text, assuming that we have no prior knowledge about virtual machine roles. We will, however assume that the administrator is at liberty to re-provision the virtual machines to different physical servers.

Virtual Machine Resources

A virtual machine can be described in two complementing ways. The first is through its static attributes which are defined in the virtual machines design. Examples are the amount of memory, number of CPUs and the placement of its filesystem locally or on a SAN. These design decisions influence the virtual machines performance and are for the most part static variables. Once the virtual machine is running, we get a series of dynamic variables describing how the virtual machine performs over time. We get the CPU consumption, network traffic, IO operations and process interrupts, to name a few.

It is easy to plan ahead for even distribution among the static resources. Memory usually dominates these decisions – there is only so much to go around. The CPU, on the other hand, can be shared, and on a multi-CPU server, virtual machines may even run along side each other without conflict. Too many virtual machines demanding CPU time at once will throttle the performance of all. But in order to avoid this, we need to have knowledge about which virtual machines are troublemakers in order to isolate them. For small sites with a convenient number of virtual machines, intuition and random observation may be sufficient for a satisfactory end-result. But what when the amount of servers and virtual machines exceeds what is practical for manual analysis? On larger sites,

the dynamic resources need to be observed as time-series variables and stored for analysis.

- **Static resources** belong to the *server* and are allocated for each VM at both boot and build time. Examples of static resources are:
 - Disk size
 - Filesystem placement
 - Memory
 - Virtual CPUs
- **Dynamic resources** are consumed by the VM at run-time. They are subject to rapid change. Examples of dynamic resources are:
 - CPU seconds
 - Network traffic
 - IO operations

The Server Redundancy Level

Uptime and service availability are strong arguments for virtualization. But this depends strongly on the infrastructures stability and capability to re-provision virtual machines. Several virtual machines on the same server means increased pressure on the server not to fail. If it would show signs of failure, like messages about a failing disk, the virtual machines must be re-provisioned as quickly as possible in order to take down the sever for maintenance. A server that crashes with virtual machines still on is a disaster which is simply described as the all-the-eggs-in-one-basket effect. There is therefore much effort required by the system administrator in order to know when and which server can be taken down.

Live migration is one of the most attractive feature when it comes to re-provisioning a virtual machine. It enables a virtual machine to move to a different server without loss in downtime. Other methods of migration exist too, like cold migration, where the virtual machine is shut down before it is moved. The latter approach usually assumes that the two servers do not share a common storage for the VMs filesystem and it has to be transported to the new server.

In both cases, however, the assumption is that there is enough capacity on the receiving servers to accommodate more virtual machines. In a multi-server environment it becomes difficult to assess the possibility of, e.g., freeing one server from all its virtual machines. Do we have enough *combined* capacity to remove one physical server? How can we find the answer to that question?

We propose a simple notation for the redundancy of server capacity called the *redundancy level*. It is " R/S " where R is the number of servers that are in use currently and S the number of which servers can be shut down and removed. The idea is to calculate the available capacity on all servers and the current usage of all the virtual machines. From this, the R/S value can be derived. It is usually enough to consider only static resources as capacity limits with most focus on

memory and disk-space if the virtual machines are stored locally.

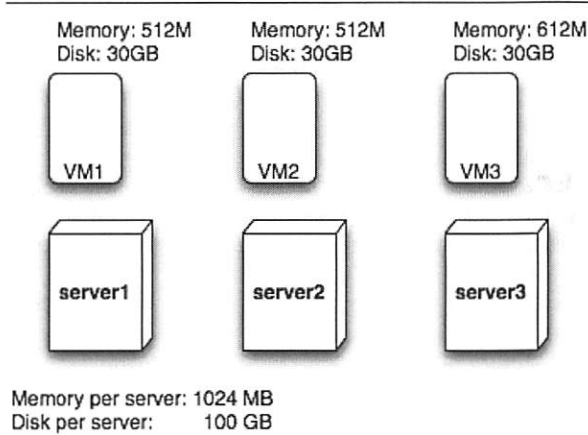


Figure 1: An example showing a server redundancy level of 3/0 because VM3 larger memory setting.

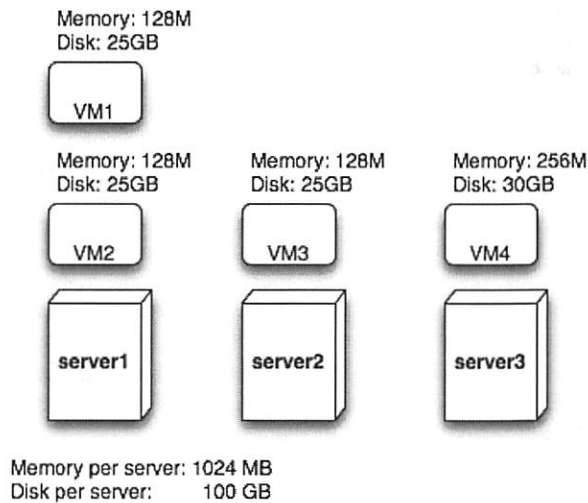


Figure 2: An example showing a server redundancy level of 3/1. With some initial planning, a level of 3/2 could have been achieved.

Example 1

In the following example we have three servers and three virtual machines: each of the three servers has 1024 MB of memory available for virtual machines as well as 100 GB storage space. VM1 and VM2 each use 512 MB of that memory together with 30 GB of storage for their harddisks. VM3 uses 612 MB of memory so that server3 has less free capacity than server2 and server1. The available memory (412 MB) is not enough to accommodate VM1 nor VM2. Server1 and server2 have both 512 MB of available capacity and cannot accommodate VM3. The result is that if server3 should be removed, we will loose VM3. The server redundancy level is 3/0 because we have three servers and no-one of them can be removed.

Clearly, if server1 goes down, then server2 would have enough spare capacity to hold VM1, but

the “S” value in the redundancy metric should be as pessimistic as possible and must therefore hold for all servers. For *only* server1 and server2 we have, in fact, 2/1.

Example 2

This example is a little bit more complicated. The servers are the same as before, but now we have more light-weight virtual machines. We see, that the total memory consumption of all four virtual machines is 640 MB, that is less than the capacity of either one of the servers. This means that a single server could accommodate all the virtual machines, which would actually result in the redundancy value of 3/2. But unfortunately, since VM4 uses 30 GB of disk space, the sum of the total disk space consumption would be 105 GB, and that is more than a single server can offer. So the reality is that the redundancy level only is 3/1.

This example shows how some initial planning could improve the systems overall redundancy level. If one server would go down, we would end up with two servers with the combined memory capacity of 2048 MB but only of 640 MB of it actually used. On top of that, the redundancy level would then be 2/0, meaning two underutilized servers where no-one can be spared.

The redundancy level mirrors the infrastructures capability to loose one or more of its servers without loss of virtual machines and is therefore valuable information to a system administrator. It can be considered as a service level policy such as “*The infrastructure is to keep a 3/1 redundancy level 90% of the time.*” It can also be useful for capacity planning, calculating how much more virtual machines of a certain type can be deployed before the redundancy level is altered. It will also show how much resources a new server should have in order to keep a certain redundancy level for planned additions of more virtual machines. The notation should not be confused with a division, where $3/1 = 3$. It is not meant to be a factor. The redundancy level of 4/2 is obviously also not the same as 2/1.

Resource Conflict Matrix

The previous analysis was from a infrastructure point of view. It did not address potential resource conflicts between the virtual machines. For this task, we propose creating matrixes for every resource type and to use graph theory for an indication of the number of resource conflicts we have. The analysis is simplest for the static variables. A resource conflict matrix is a square diagonal matrix with all the virtual machines. For each virtual machine pair, we put a 1 in their positions if they have a resource conflict for that particular resource type, otherwise a 0.

Let’s consider filesystem placement as a static resource (i.e., not current disk usage). Two virtual machines have a conflict if their filesystem is placed

on the same harddrive. This would in principle imply, that two virtual machines placed on the same SAN are in conflict even if they run on different servers. These conflicts do in other words not imply that the performance is going to be poor. It simply states that there is a potential between those two virtual machines that they may influence each other.

The resource conflict matrixes are of size $V \times V$ where V is the number of virtual machines on the infrastructure. One technique to compress the matrix into a single value is to treat the matrix as an adjacency matrix for a bi-directional graph, and to calculate its connectivity using the formula:

$$\frac{C}{\frac{1}{2} V(V-1)}$$

The connectivity of a graph is the number of links divided by the possible number of links. In this case the number of conflicts C divided by the possible maximum number of conflicts. Its highest value, 1, would imply that there is a conflict between all virtual machines for that particular variable. A zero, 0, would mean the opposite. For each variable one can therefore easily get a value describing its current level of conflicts.

A simple example of the analysis is given in Figure 3. The same case would be for the static CPU

	VM1	VM2	VM3	VM4
VM1	0	1	0	0
VM2	1	0	0	0
VM3	0	0	0	0
VM4	0	0	0	0

(a) With all servers up. Conflict rate $1/6 = 0.1667$.

	VM1	VM2	VM3	VM4
VM1	0	1	0	0
VM2	1	0	0	0
VM3	0	0	0	1
VM4	0	0	1	0

(b) With server2 down and VM3 re-provisioned to server3. Conflict rate $2/6 = 0.333$.

	VM1	VM2	VM3	VM4
VM1	0	1	1	0
VM2	1	0	1	0
VM3	1	1	0	0
VM4	0	0	0	0

(c) With server2 down and VM3 re-provisioned to server1. Conflict rate $3/6 = 0.5$.

Figure 3: A resource conflict matrix for filesystem placement based on Example 2. As long as all servers are up, we have a low rate of conflicts. If server2 goes down (matrix b and c), the conflicts increase depending on how the virtual machines are re-provisioned. b) is a better solution than c) because it has the lowest conflict rate.

resource if two virtual machines on the same single-CPU server would be in conflict. Few virtual machines, as used in our examples, may produce obvious matrixes. On large sites, however, the resulting matrix may become too large for manual review.

For the dynamic variables, such as CPU or IO usage, we need to compare the actual behavior of the virtual machines. If a time-series profile of each virtual machine existed, it could be compared to see if two virtual machines historically tend to demand the shared resources at the same time. If so, they are in a conflict. The level of correlation or probability of conflict between the two makes for a more fuzzy description than 1 or 0. One can still calculate the average probability rate between all the virtual machines, however it does not carry the same interpretation as the graph connectivity mentioned above, since the result would not represent the average number of conflicts anymore, but the average conflict probability. Other analysis methods could also be applied to the matrix, such as Principle Component Analysis or centrality, however the interpretation of these results in this case are still being studied.

The resource conflict matrixes are a resource centric description of the infrastructure's state. Connectivity or average conflict levels are ways to compress information into more usable formats. Some level of intuition and knowledge is still needed to interpret its results. It can highlight uneven distribution of resources where there should be none. A matrix is beneficial for a graphical representation and can contain more information than only an average.

We also need a virtual machine-centric perspective of the infrastructure in case we investigate a particular virtual machine or want to make the best provisioning decision for a new one.

Location Conflict Table

The previous section showed how we can get an overall view of the resource conflicts and that re-provisioning of virtual machines influences the result based on what server it is placed on. The example was only for a single resource. How could we get an impression of the conflicts concerning all resources in order to make the best decision? Our solution to this problem is to list the available servers based on the resulting conflicts for all resources if that would be its location. Let us consider the case above where server2 goes down and VM3 needs to be moved. The resulting location conflict table is shown in Table 3.

Location conflict table for VM3			
Location	Filesystem		
	Placement	CPU	Total
server1	2	2	4
server3	1	1	2

Table 3: An example location conflict table showing server3 with less conflicts than server1.

For servers with equal performance, one should choose the one with the least resource conflicts. In many cases, some conflicts are unavoidable. Some conflicts may then be given more importance, such as those based on time-series profiles.

The information in this table will potentially change for all virtual machines every time a virtual machine is re-provisioned. This means that if server2 now contained two virtual machines, we would have to make a decision about the first and then the second as if the first has already moved.

Implementation Into MLN

The methods mentioned above are general ways to analyze the state of the infrastructure. The benefit with quantifiable metrics is that their calculation can be automated. The service redundancy level, resource conflict matrixes and location conflict table are implemented into MLN for static variables with preliminary support for dynamic variables also. MLN has enough knowledge about virtual machine location and resources to determine conflicts automatically.

The benefit of this information is currently studied in a master thesis with regard to automated re-provisioning and analysis [13]. MLN is only capable of providing current state values so it cannot compute "what if"-results for planning yet. Work is under way to include analysis features where the user could propose changes to the provisioning or other virtual machine metrics and see the result before they are committed.

Discussion

The server redundancy level only assumes that we will re-provision virtual machines from a given server onto the others. It does not assume that we could re-arrange all the virtual machines optimally on all server for the highest utilization. We have identified some additional challenges in that case which are still under consideration. The most important point is what strategy to use for the re-provisioning process. We have identified a few alternatives:

- **Least Migrations** This would reduce the risk of virtual machines dying in the migration process.
- **Least Memory Copied** Migration time depends on the main memory of the VM to be copied in the background. Less memory to copy would result in faster migration.
- **Minimize resource conflicts** Never-mind the number of migrations, just reduce the number of conflicts to a minimum.
- **Most Important Last** Some VMs are more important than others. Avoid touching the most important.

Which strategy is best depends on the context. In an automated re-provisioning scenario, the user should be able to express to the system which strategy is preferred.

A resource conflict is not equal performance degradation, but it may be a good indicator of where to look for answers if the virtual machine performs poorly. The resource conflict matrixes can be used to look up the current conflicts of a given virtual machine at any time. We consider this helpful support information for system administrators.

Some of the resources and conflicts are diffuse. Consider five single-CPU virtual machines on a four-CPU server. Who is in conflict with who? The strictest interpretation is that all five are in conflict with each other.

The connectivity or average conflict rate may not carry much information at the first calculation. The optimal or lowest possible value is entirely context dependent. It is therefore not suitable to be used as a comparison between two different infrastructures. However it is valuable as a measurement at re-provisioning. It is a way to observe how an addition of a new or existing virtual machine influences the rest.

Conclusions and Future Work

Virtual machines can easily be deployed using MLN and have acceptable performance levels through the Xen virtual machine framework. The ideal scenario would be one virtual machine on each server, but that is not always possible. Finding the best placement of the virtual machines in order to avoid resource conflicts is an open challenge.

Three methods were proposed for infrastructure analysis based on our own experience. Each method is technology independent and can be applied by hand for fair-sized infrastructures. They are also implemented in MLN in order to cope with larger scenarios where manual calculation is impractical.

We recognize that these methods have issues of their own but we see this as steps towards a better understanding of how to manage virtual machines successfully. This research field is only just emerging as more start to use virtual machines on a large scale and encounter the same kind of problems.

Work on MLN will continue in this direction and may function as a way to let the industry test our ideas and provide us with valuable feedback. Users should also be able to define other resources which are included into MLN's existing analysis. This may be easy to accommodate for static variables, however the dynamic ones would also imply that a monitoring framework of that variable should exist.

Related Work

In recent years, many researchers have promoted the use of virtualization and virtual machine technologies in scientific computing and HPC environments. Such arguments appeared initially in the literature of grid computing. For example, Figueiredo and coworkers [6] proposed using virtual machine technology in combination with management middleware to simplify

the task of seamlessly providing distributed computing resources to grid end users. They noted that using virtualization provided several advantages over grids constructed via real systems. Virtualization also offers many advantages to high performance computing; for a succinct summary, see Mergen, et al. [5].

Several groups have performed performance analyses of scientific problems in virtual environments. Figueiredo and coworkers [6] compared the performance of their virtualization-based approach with that of the regular operating system by running a few of the benchmarks from the SPEC suite directly on a Linux system and in a virtual machine running the same operating system under VMWare. Their results indicated that only that minimal overhead was associated with employing virtualization, in the range of about 2-4%. They also considered the time required for virtual machine setup, either via the full startup process (i.e., VM reboot) or by restoring a saved VM. Startup times generally ranged from about 30 seconds to 1 minute, depending of the specifics of the startup method and VM disk file storage location.

Youseff and coworkers [10] compared the performance of Red Hat Enterprise Linux running directly on the hardware as well as a guest operating system under the Xen environment via a series of standard benchmark applications. Their tests included three categories of calculation: micro-benchmarks each focused on a single system resource (CPU, network communication, memory and disk I/O), a series of matrix-based computations designed to test parallel program execution efficiency, and a single scientific simulation taken from the HPC Challenge benchmark suite (the MIT GCM exp2 which models a planetary ocean circulation process). In all cases, including the latter, these researchers found no statistically significant performance degradation from employing virtualization.

Bjerke [14] implemented Xen virtualization for HPC on Itanium systems. He presents some performance data for typical software building processes, again finding little difference between the native system and Xen virtual machine instances.

Finally, Vogels has explored virtualization for HPC in the Windows environment/.NET framework [15]. He ran benchmarks from the SciMark suite, focusing on comparing different virtual environments. In general, however, his results indicate that virtualization is a viable alternative for Java-based high performance computing in this environment as well.

Previous work on checkpointing has focused on general solutions for applications on UNIX systems. For example, Plank and coworker [7] created the libckpt library with the goal of rollback recovery for an executing program on UNIX systems. The library worked by periodically saving the application's current state to a disk file. In the event of a failure (i.e.,

program crash), the program could be restarted from the most recent save point (checkpoint). The facility was capable of operating in a fully automated mode, without requiring any program modifications, for existing applications; programmers could also add checkpointing-related directives to the code if the source code was available. The former is most directly comparable to virtualization. The researchers tested their approach using several computationally-intensive benchmarks. For these applications, the overhead associated with using libckpt in fully automated mode was considerable, ranging from about 5% to about 15%, with the larger values associated with the larger and most realistic benchmarks.

Wang and coworkers [8] performed similar work at about the same time. Their libckpt library for UNIX systems similarly periodically saved the execution state from user applications in a transparent manner. Their work also included a generalization feature which allowed a checkpoint to be reused with different input/data.

VM management has also received substantial research attention. The closest work to MLN is probably the In-VIGO system of Adabala and workers [16]. It is an example of using virtualization for grid computing, specifically by constructing virtual grids on top of real systems via a software middleware layer. In this way, In-VIGO provides grid computing environment in which the actual physical systems and resources are transparent to grid users. It is designed to both simplify using the grid computing resources for end users as well as to simplify the grid management tasks for system administrators.

However, unlike MLN, it is a quite complex system (albeit a powerful one) necessitating a significant learning curve. In addition, many of its features are simply not needed for production scientific computing. The follow-on work, the VMplants facility [17], provides a facility that is similar to MLN. It provides virtual machine creation, shutdown and cloning. Its operation is client-driven, in keeping with the goals of In-VIGO of simplifying the end user experience, contrast to MLN, which can employ either a push or pull approach.

Finally, Clark and coworkers [9] have studied migrating running virtual machines across physical hosts without the need for even temporary suspension. They found that the challenges of migrating live memory to be challenging, especially for applications with intensive memory write rates. They offer a solution for VM live migration within a group of discrete systems or a cluster with high performance network interconnects and network-based disk storage.

Computation Details

All Gaussian 03 [11] jobs were limited to 256 MB of memory and used the 32-bit version of the

program. The VMs used were allocated 512 MB of memory. The key components of the computer configurations were as follows: 1.8 GHz Xeon with 1 GB main memory (master node for parallel jobs); 2.8 GHz Xeon with 2 GB main memory; 2 GHz AMD-64 with 1 GB main memory. The network interconnect was 100BaseT. Parallel jobs made use of the Linda parallel execution environment [12], as required by the application. Job details: (1) HF/3-21G Opt Freq on Buckminsterfullerene, 540 basis functions; (2) B3LYP/3-21G Force SCF=NoVarAcc on Valinomycin, 882 basis functions; (3) MP2/6-311+G(2d,2p) Opt Freq on Malonaldehyde, 171 basis functions. All jobs were run on otherwise idle systems.

The following table gives the raw results for these jobs. The column headings have the following meanings: RHEL=job run on system running standard kernel; Xen=job run on system running Xen-enabled kernel, but not in a VM; VM=job run in a Xen VM (RHEL guest OS). The single processor jobs were all run on the slowest node, which also served as the master node for the parallel jobs.

Job	Elapsed Time (seconds)				
	Single Processor		Parallel Execution		
	RHEL	Xen	VM	Xen*3	VM*3
1	346	359	348	169	168
2	727	755	733	268	265
3	420	433	453	366	338

Table 4: Raw results as elapsed time for each test scenario.

Author Biographies

Kyrre Begnum is currently completing his Ph.D. in Network and System Administration at Oslo University College in Norway. Part of his research focuses on managing virtual infrastructures, and he is the author of the Manage Large Networks (MLN) VM administrative tool.

Æleen Frisch has been a system administrator for over 20 years. She currently looks after a pathologically heterogeneous network of UNIX and Windows systems. She is the author of several books, including Essential System Administration (now in its third edition). Æleen was the program committee chair for LISA '03 and is a frequent presenter at USENIX and SAGE events, as well as presenting classes for universities and corporations worldwide.

Ingard Mevåg is a graduate from Oslo University College with a masters degree in Network and System Administration. After part-time work at Game Index Statistics & Analysis AS during the education, he is now employed at ABC Startside AS as a system administrator.

Matthew Disney has been working in systems administration since 1998. He has a BS in Computer Science from the University of Tennessee and an MS in

Network and System Administration from the University of Oslo. He is currently working as a cyber security administrator at Oak Ridge National Laboratory.

Bibliography

- [1] Begnum, K. and J. Sechrest, *The MLN Home Page*, <http://mln.sourceforge.net>; Last accessed August 28, 2006.
- [2] Begnum, K., "Manage Large Networks of Virtual Machines," *Proceedings of the 20th Large Installation System Administration Conference*, USENIX, 2006.
- [3] Begnum, K. and M. Disney, "Scalable Deployment and Configuration of High-Performance Virtual Clusters," *CISE/CGCS 2006: 3rd International Conference on Cluster and Grid Computing Systems*, 2006.
- [4] Begnum, K., K. Koymans, A. Krap, and J. Sechrest, "Using Virtual Machines for System and Network Administration Education," *Proceedings of SANE conference*, 2004.
- [5] Mergen, Mark F., Volkmar Uhlig, Orran Krieger, and Jimi Xenidis, "Virtualization for High-Performance Computing," *SIGOPS Operating Systems Review*, Vol. 40, Num. 2, pp. 8-11, 2006.
- [6] Figueiredo, Renato J., Peter A. Dinda, and José A. B. Fortes, "A Case For Grid Computing On Virtual Machines," *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, p. 550, IEEE Computer Society, Washington, DC, 2003.
- [7] Plank, James S., Micah Beck, Gerry Kingsley, and Kai Li, "Libckpt: Transparent Checkpointing under Unix," *Proceedings of USENIX Winter 1995 Technical Conference*, pp. 213-224, New Orleans, Louisiana, January, 1995.
- [8] Wang, Yi-Min, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala, "Checkpointing and Its Applications," *fics*, 00:0022, 1995.
- [9] Clark, Christopher, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, "Live Migration of Virtual Machines," *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 273-286, Boston, MA, May, 2005.
- [10] Youseff, Lamia, Rich Wolski, Brent Gorda, and Chandra Krintz, "Paravirtualization for HPC Systems," Technical Report 2006-10, University of California, Santa Barbara and Lawrence Livermore National Lab, 2006.
- [11] Frisch, M. J., G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, J. A. Montgomery Jr., T. Vreven, K. N. Kudin, J. C. Burant, J. M. Millam, S. S. Iyengar, J. Tomasi, V. Barone, B. Mennucci, M. Cossi, G. Scalmani, N. Rega, G. A. Petersson, H. Nakatsuji, M. Hada, M.

- Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, M. Klene, X. Li, J. E. Knox, H. P. Hratchian, J. B. Cross, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, P. Y. Ayala, K. Morokuma, G. A. Voth, P. Salvador, J. J. Dannenberg, V. G. Zakrzewski, S. Dapprich, A. D. Daniels, M. C. Strain, O. Farkas, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. V. Ortiz, Q. Cui, A. G. Baboul, S. Clifford, J. Cioslowski, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, C. Gonzalez, and J. A. Pople. *Gaussian 03, Revision D.02*, Gaussian, Inc., Wallingford, CT, 2004.
- [12] Scientific Computing Associates Inc., *The Linda Home Page*, <http://www.lindaspaces.com>.
- [13] Ingard Mevåg, "Towards Automatic Management and Live Migration of Virtual Machines," Master's thesis, University of Oslo, Oslo University College, (UiO / HiO), Oslo, Norway, May 2007.
- [14] Bjerke, Håvard K. F., "HPC Virtualization with Xen on Itanium," Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, July 2005.
- [15] Vogels, Werner, "HPC.NET – are CLI-based Virtual Machines Suitable for High Performance Computing?," *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, p. 36, IEEE Computer Society, Washington, DC, 2003.
- [16] Adabala, Sumalatha, Vineet Chadha, Puneet Chawla, Renato Figueiredo, José Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu, "From Virtualized Resources to Virtual Computing Grids: The In-VIGO System," *Future Generation Computing Systems*, Vol. 21, Num. 6, pp. 896-909, 2005.
- [17] Krsul, Ivan, Arijit Ganguly, Jian Zhang, José A. B. Fortes, and Renato J. Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing," *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, p. 7, IEEE Computer Society, Washington, DC, 2004.

OS Circular: Internet Client for Reference

Kuniyasu Suzuki, Toshiki Yagi, Kengo Iijima, and Nguyen Anh Quynh
– National Institute of Advanced Industrial Science and Technology, Japan

ABSTRACT

OS Circular is a framework for Internet Disk Image Distribution of software for virtual machines, those which offer a “virtualized” common PC environment on any PC. OS images are obtained via the stackable virtual disk “Trusted HTTP-FUSE CLOOP”.

The system is designed to utilize Mirror servers and Proxies for highly-scalable worldwide deployment. OS Circular easily and efficiently handles both partial and periodic OS updates, including a rollback facility to ease experimentation with new OS images that might not be ready for production. This paper describes the design of OS Circular and the techniques to reduce the network traffic for quick downloading and booting.

Introduction

We have developed “OS Circular” [1, 2], a technique for booting any OS on an anonymous PC over the internet. It enables selection of OS images and can facilitate a reference installation. This means that an OS image can be “installed” to ease testing and feasibility testing of new functionality before it is installed on the local hard drive for production use. The previous (reference) version of an OS image stores old application software and enables opening of files in any previously supported format. In a world of frequent security updates, this environment dramatically enhances the ease of testing OS service packs.

Virtual machines host a common PC environment on a standard PC. Software to host virtual machines is easy to obtain and use, with open source packages QEMU, KQEMU, KVM, and Xen in addition to free versions of VMware and Virtual Box. Recent performance of virtual machines has quite low overhead and enables use of guest OSes without high resource utilization. Furthermore, newer x86 architecture CPUs have a virtualization extension (Intel’s VT or AMD’s SVM) that promotes even better virtual machine software (including a mode for trapping sensitive instructions). Both KVM and Xen-HVM use this virtualization extension and offer full virtualization.

One current challenge for virtual machines is the scheme used to share virtual disks on the Internet [3, 4]. These virtual disks represent both OSes and application software and should be updated periodically (e.g., for security). The previous disk image should be archived efficiently and might be used for replaying the old OS image when an update is unsatisfactory. New master disk images could be shared with a potentially massive number of users, and efficiently doing that is one of the primary goals of OS Circular.

OS Circular is designed to utilize existing infrastructures in order to enable global deployment and

minimize maintenance cost. As a client-centric system, OS Circular reduces server load by limiting server functionality to file distribution. OS Circular uses HTTP for file distribution because of the ease of accessing Web hosting services (including exploitation of mirror servers and cache proxies).

The client PC checks data for validity during downloads (which are provided by Trusted HTTP-FUSE CLOOP, a stackable virtual disk). Trusted HTTP-FUSE CLOOP includes support for update of virtual disks. OS Circular uses full virtualization (to avoid exploits due to the kernel of the guest OS being insecure). It also includes an automatic security update service.

This paper details OS Circular’s design and implementation paradigm. We mention related work, then describe the virtual machine as an abstraction layer. The next section describes the requirement of virtual disks. Subsequently, we discuss details of the Trusted HTTP-FUSE CLOOP and the current implementation and performance of OS Circular. Finally, we discuss future plans and conclude.

Related Work

Virtual disks are a popular means for distribution of ready-to-use OS images for virtual machines. The OS Zoo project [5] distributes many virtual disk files for QEMU (an open source machine emulator and virtualizer). This eases experimentation with various operating systems since installation is relatively simple. However, virtual disks often must be treated in some ways as a single – potentially extremely large – file. Downloading hundreds of megabytes can take quite a while. Furthermore, even the update of a single bit requires the (time-consuming) reconstruction of the entire virtual disk, a real detriment when an urgent security update is required.

FLOZ (Free Live OS Zoo) [6] is an interesting derivative work of OS Zoo project. It enables booting

of many OSES via a Web browser. FLOZ runs a QEMU virtual machine on the server and transfers the visual console of the QEMU to a Web browser on some client. While very innovative for OS testing, performance and scalability are limited due to its server-centric design which also suffers greatly as Internet network latency (for file access) increases. Furthermore, OSES hosted on FLOZ do not allow Internet connections due to server network resource and security concerns.

“Collective” [3] and “Ventana” [4] also propose client-centric systems. They run virtual machines on a client and download “diffs” of updated OS images.

- Collective uses VMware’s COW (Copy On Write) feature for partial update. This is a disk block level mapping and thus handles any filesystem format. Updated data is saved to a file, making it easy to deal with. Unfortunately, it is difficult to map many COW images. The developers established the company called Moka5 to offer LivePC [7], an improved version of Collective.
- Ventana is a virtualization-aware filesystem with versioning, access control, and disconnected operation. It is a management system for virtual disks and offers a customized view of a disk image for each user. Unfortunately, it is based on NFS and it is not designed for a massive number of anonymous users on the Internet.

Virtual Machines

Full virtualization offers a sort of abstraction layer for OSES that provides a common virtualized PC environment on an anonymous PC. We can install a guest OS with its normal installer, update the OS with its usual package management software, and migrate it to other PCs via virtual disks. Various virtualizers do this in different ways: VMware is used on SoulPad [8], VAT (Virtual Appliance Transceiver) of Collective [3], and Internet Suspend/Resume [9].

Device Model

It is critical that full virtualization offer the same device model that a guest OS expects. When this virtualization is present, the guest OS need only prepare drivers for abstracted devices – not for any specific model or type of disk.

QEMU-DM (Device Model) is becoming popular in the open source community. It assumes RealTek RTL8029 for NIC, Cirrus Logic GD5446 for Video Card and a few others. Both Xen and KVM also support QEMU-DM and guest OS only have to support them.

The differences among IA32 architectures cause no problems for the emulators because recent OSES offer common i386 packages for IA32 which run on Pentium Pro or later IA32 CPUs. QEMU, KQEMU and KVM offer the Pentium II architecture for guest

OSes but Xen-HVM offers the same architecture as the real CPU upon which it is running. These differences are mitigated by the “universal” packages of OSES.

VM Loader

OS Circular requires both a virtual machine and a stackable virtual disk. For convenience, we have developed and offer a 1CD Linux installation which includes both of these.

This host OS supports the drivers for real devices. SoulPad and VAT of Collective use KNOPPIX as the host OS, because KNOPPIX has an “AutoConfig” function that automatically detects available devices at boot time and loads the appropriate drivers. The combination of AutoConfig and full virtualization of the host OS acts as a virtual machine loader.

We offer VMKNOPPIX, a collection of virtual machine software on KNOPPIX, as a VM Loader. It includes QEMU, KQEMU, KVM, and Xen-HVM, each of which offers full virtualization. Xen-HVM and KVM require a CPU with the virtualization extension (Intel VT or AMD-SVM) but QEMU and KQEMU run on any IA32 architecture. KNOPPIX acts as the host OS and has drivers for almost any PC.

Requirements for Virtual Disks

Virtual disks should offer features such as versioning, globalization, and security; see [4] for the main requirements. These requirements motivate the design for Trusted HTTP-FUSE LOOP.

Versioning

Versioning of virtual disks is very desirable:

- non-persistent versioning is used for “undo” of operations
- while persistent versioning is used for “roll-back” of OS image

Versioning is also important for sharing and customization of virtual disks.

Some virtual machine software supports the “undo” function, including the “non-persistent” mode of VMware and “CopyOnWrite” of QEMU and User-ModeLinux. Xen uses DeviceMapper of Linux for non-persistent versioning. Virtual Disks should offer persistent versioning.

Trusted HTTP-FUSE CLOOP has the same versioning scheme as Venti [10], Plan9’s archival storage system that permanently stores data blocks (blocks which comprise the data of a filesystem – the filesystem’s structure is independent of the block storage). Each block is stored in a file whose name is its SHA1 hash. The system enforces a write-once policy since no other data block should ever have the same hash. Duplicate data is easily identified (since it has the same hash) and a given data block is stored only once. Data blocks cannot be removed, making the system ideal for permanent or backup storage. Unfortunately, Venti requires a special protocol for accessing the filesystem, and this limits its scalability.

Trusted HTTP-FUSE CLOOP saves data blocks in much the same way as Plan9 and utilizes HTTP to distribute them.

Globalization

One goal of OS Circular is the provision of virtual disks that can be shared via Internet. This does not connote the complete downloading of a virtual disk file but rather access to any part of a disk as requested.

The “Remote Block Device” scheme meets this requirement and is found in many implementations, including iSCSI, AOE (ATA over Ethernet), iFCP, and more. Unfortunately, all of these use special protocols and require special daemons on the server. Most of them are designed for use on a high speed LAN and aren’t as useful on the slower Internet. Trusted HTTP-FUSE CLOOP re-constructs a virtual disk with the partial block files which are downloadable via HTTP.

Virtual disks also require a “disconnect operation” to achieve Mobile Computing. The “AFS” and “Coda” filesystems deal with the disconnect operation but also require special protocol and daemons for servers. Stateless Linux [11] offers a disconnect operation for a Thin Client PC. Stateless Linux runs with network storage or snapshot image saved local storage. Block files of Trusted HTTP-FUSE CLOOP are also saved to a local storage and re-usable (when whole block files are saved in a local storage, a network connection is not required).

Security

Basically, security management is independent of the virtual disk implementation. Security of the kernel and applications should be managed by security software or package manager. A virtual disk makes only

the commitment to keep the integrity of its contents. Probably the biggest part of the security for a network virtual disk is the prevention of intrusion. One way to prevent such an intrusion is the use of secure communication, but this requires a fixed server which limits scalability. Trusted HTTP-FUSE CLOOP adopts a client-centric contents validation mechanism with a driver that checks the validity of block contents when mapped to the virtual disk. It allows distributing block data by anonymous servers.

Trusted HTTP-FUSE CLOOP

KNOPPIX’s CLOOP (Compressed Loop back device) spawned the big picture idea for HTTP-FUSE CLOOP’s filesystem. KNOPPIX saves block device data to a file in order to reduce disk consumption (though a CLOOP file is still big). Traditional CD-KNOPPIX requires about 700 MB and must be treated as one file, slowing downloads. When even just a single bit is updated, a big CLOOP file must be rebuilt. Furthermore, CLOOP has no security protection.

To mitigate these problems, we adopted the block management style of Venti [10]. Data on a block device is partitioned into data blocks of a fixed size, compressed, and saved. Block files are treated as network transparent between local and remote machines, with local storage acting as a cache. The downloaded block files are validated by their SHA1 hash value. This yields these features:

- A block file is made of split, compressed block device blocks (default size is 256 KB) blocks (The original CLOOP’s split block size was 64 KB which was too small and created too many files).

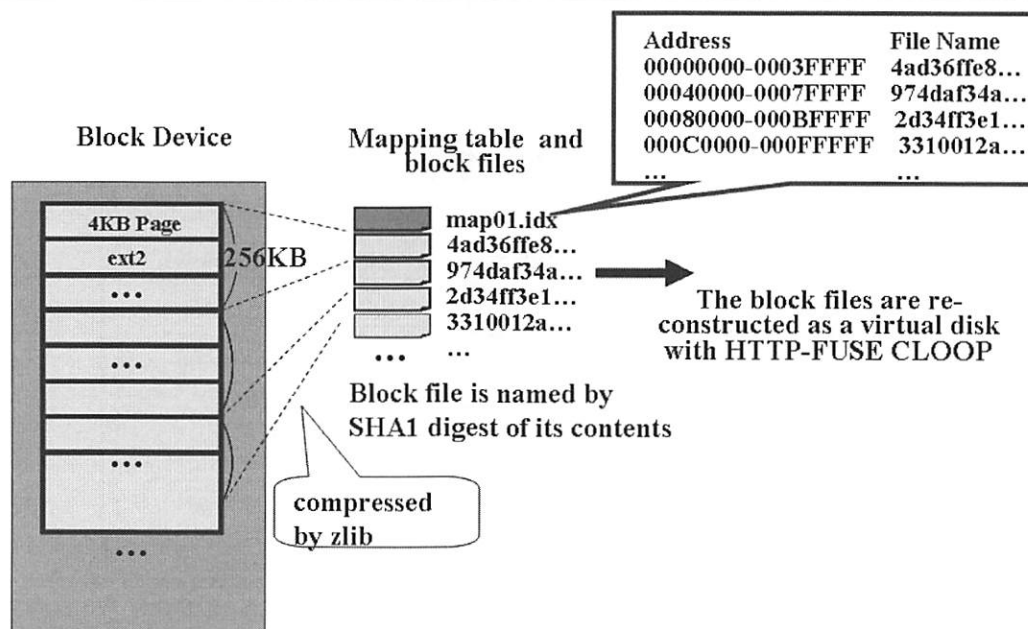


Figure 1: Creation of block files from OS image.

- Block files are mapped to a loopback device with a mapping table file.
- This mapping for block files is performed when a relevant read request is issued. After mapping, the block file is erasable from local (cache) storage, since it can be downloaded anew any time it is needed.
- The name of a given block file is the value of its SHA1 hash with all the good properties listed above.
- Block files are downloaded from the HTTP server because HTTP is expected to be strong file delivery infrastructure as demonstrated by mirror servers and proxy servers.
- A proxy server has a size limitation on cached file sizes so block files should be smaller than this size.
- When mapping a block file to the loopback device, the block's contents are hashed into a SHA1 file name which is listed in the mapping table file.
- The block device is Partially Updatable: When an application is updated on the original block device, relevant block files and the mapping file are renewed; cache block files related to the non-updated block are reusable.
- "FUSE" (File system in USER-space) [12] is used to implement the virtual loopback device.

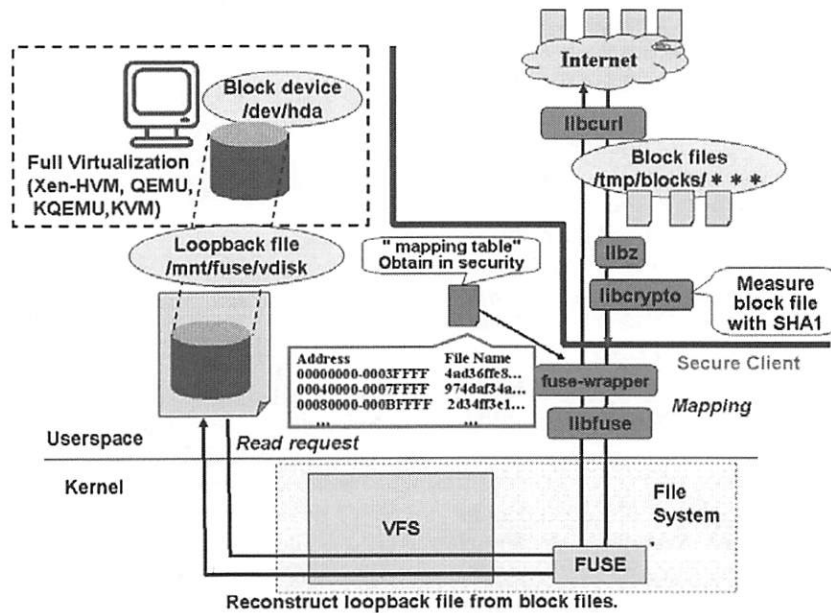


Figure 2: Diagram of Trusted HTTP-FUSE CLOOP.

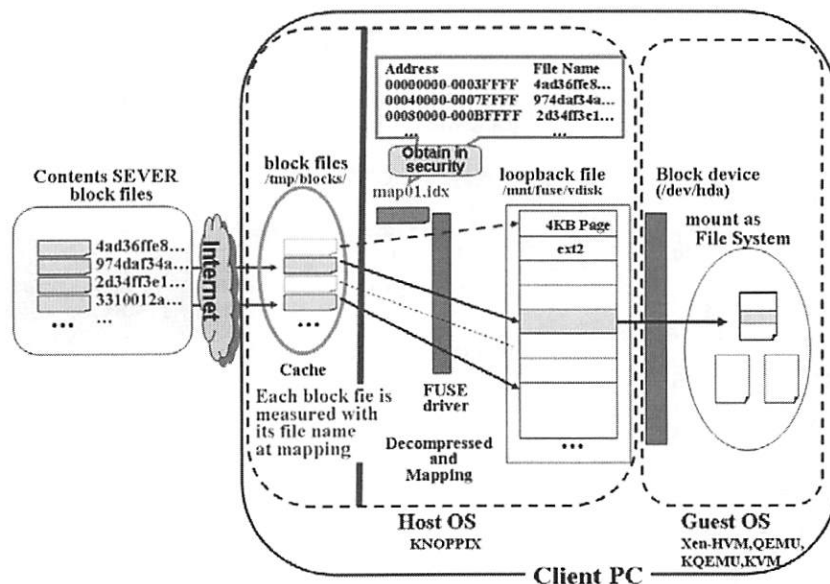


Figure 3: Block mapping of Trusted HTTP-FUSE CLOOP.

Figure 1 depicts the creation of block files and the mapping table file “map01.idx” which is also made from a block device which includes root filesystem. Each block file is named by its SHA1 hash.

Figure 2 shows the block diagram of Trusted HTTP-FUSE CLOOP. The loopback device is mapped as a normal block device on a virtual machine. The main program is implemented as a part of FUSE wrapper program. A maintains the validity of the virtual block and must be distributed in secure way. The mapping table file is used to set up Trusted HTTP-FUSE CLOOP. When a read request is issued, Trusted HTTP-FUSE CLOOP driver searches the relevant block file using the mapping table. When the relevant file exists on a local storage, that file is used; otherwise, the file is downloaded from Internet.

Block file elements are downloaded from an HTTP server with “libcurl”, the Client for URLs library. Each downloaded file is decompressed by “libz”, hashed by “libcrypto”, and logged into /var/log/fs_wrapper_PID.log. Invalid file transfers (due to network errors or security breaches) are detected using the SHA1 hash. Figure 4 shows an example that detects a defective block file. The downloaded block file is first stored at local storage. If the storage space is insufficient (more than 80% used), previously downloaded files are removed by LIFO (a water mark algorithm). Figure 3 shows Trusted HTTP-FUSE CLOOP from the viewpoint of block mapping.

Partial Update By Adding Block Files

The mapping table handles block addressing. Any update of HTTP-FUSE CLOOP is performed by adding updates both to the block files and to the mapping table.

Unchanged (local) block files become reusable for caching. To achieve this function, the HTTP-FUSE CLOOP filesystem treats block-unit update as an EXT2 filesystem. The “iso9660” filesystem turns to be unsuitable for HTTP-FUSE because partial update of an iso9660 file changes the location of subsequent blocks. As usual, updated blocks are saved to a file named by the SHA1 hash of the block. Collision of file name is extremely rare (this being the goal of the SHA1 hash). In the unusual event of a collision, we can check and repair the problem before uploading the block files.

Figure 5 shows an example of an HTTP-FUSE CLOOP update which creates a new mapping table file “map02.idx” and associated block files. This is particularly useful when updating KNOPPIX applications, especially in the case of security updates. Furthermore, we can rollback to an old filesystem by reinstating the previous mapping table “map01.idx” and block files.

Optimization for Download

Trusted HTTP-FUSE CLOOP is sensitive to network latency because small block files are downloaded in the order the blocks are requested.

Our original implementation suffered performance problems at boot time until we implemented two new functions: “netselect” and “DLAHEAD” (download ahead).

The “netselect” functionality searches for the lowest latency download site among candidates (through judicious use of “ping”). We arranged the HTTP sites to be dispersed across across the global Internet, a spread that also foster load-balancing of HTTP services.

```
1150452051.109: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :missed.
1150452051.112: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :hits.
1150452051.112: #00000001(166cbaedbb1cc836e7c95d7d9943efde5a53829e) :missed.
1150452051.113: #00000002(29c4e363dbad648072751calf856e5780dd2981d) :missed.
1150452051.114: #00000003(fa8ad05b713a9cf8a701636ca6c353dc58fd6b6fd) :missed.
1150452051.114: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :missed.
1150452051.128: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :hits.
1150452051.128: #00000005(916f62a6e2caedc1279a0a74975a406ddb60ec25) :missed.
1150452051.129: #00000006(19111dfc877a4fe241e125d10176d85a99b4bb86) :missed.
1150452051.130: #00000007(950c1d7623b374f8e03309a93041f5adfa3af80f) :missed.
1150452051.130: #00000008(486472b0ee27157d755bd59d623179cfc0034747) :missed.
```

Figure 4a: Correct downloading of block files.

```
1150452375.989: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :missed.
1150452375.993: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :hits.
1150452375.993: #00000001(166cbaedbb1cc836e7c95d7d9943efde5a53829e) :missed.
1150452375.994: #00000002(29c4e363dbad648072751calf856e5780dd2981d) :missed.
1150452375.995: #00000003(fa8ad05b713a9cf8a701636ca6c353dc58fd6b6fd) :missed.
1150452375.996: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :missed.
1150452375.997: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :hits.
1150452375.997: #00000005(916f62a6e2caedc1279a0a74975a406ddb60ec25) :missed.
1150452375.998: #00000006(19111dfc877a4fe241e125d10176d85a99b4bb86) :missed.
E: can't validate block.
```

Figure 4b: Invalid block file is detected.

Figure 4: Log of Trusted HTTP-FUSE CLOOP (/var/log/fs_wrapper_PID.log). The “missed” tag indicates a block requires downloading; “hits” indicates the block file is in local storage.

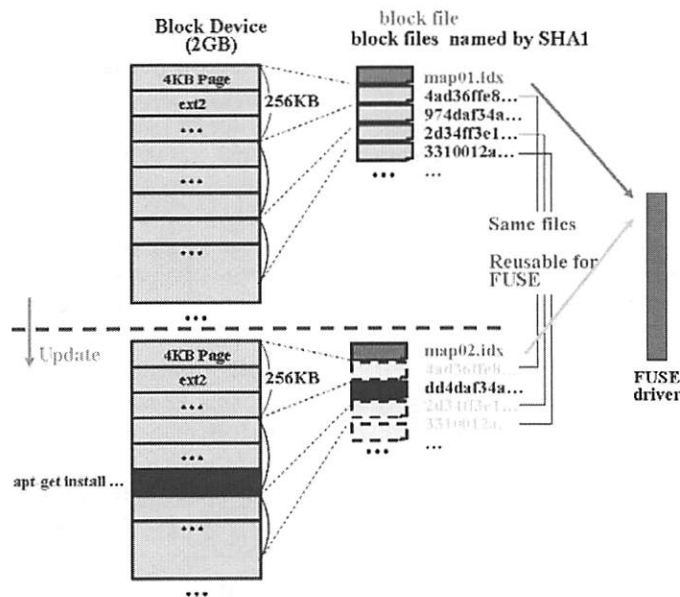


Figure 5: Update of HTTP-FUSE CLOOP.

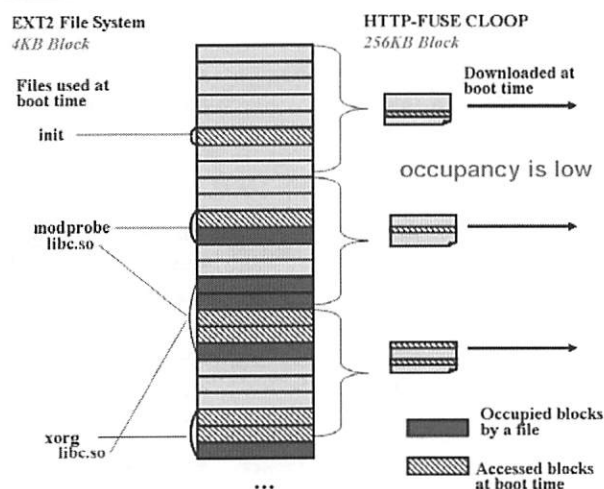


Figure 6: Ext2 filesystem is translated to block files of HTTP-FUSE CLOOP and the occupancy of accessed 4 KB data blocks at boot time.

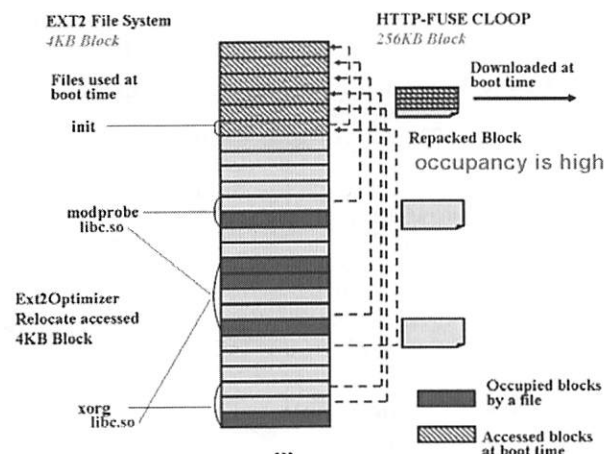


Figure 7: Ext2optimizer repacks the data backs to be formed in line.

DLAHEAD downloads the block files necessary for boot, potentially before they are requested, from a list of required block files contained in the boot profile. While this profile depends on the particular client PC, the difference among varying boot sequences is small with the greatest variation caused by the device drivers themselves. DLAHEAD establishes multiple (default: four) connections in order to download block files in parallel and thus reduces apparent network latency. Downloaded block files are saved to a local storage and work as a cache.

File System Optimization

Any block size mismatch between filesystem and virtual block device causes fragmentation. For example, EXT2 filesystems generally assume a 4 KB blocksize while HTTP-FUSE CLOOP assumes blocks of size 256KB. Even if a single block (4 KB) of a local file is requested, HTTP-FUSE CLOOP downloads the relevant (compressed) 256KB block file and decompresses it. If the next read request isn't included in some already downloaded block files, HTTP-FUSE CLOOP must download and decompress the another block file.

A typical boot sequence and subsequent system operation requires much smaller size blocks than the downloaded block files. This level of "occupancy" depends on the filesystem type, the size of the entire filesystem, the block size, and the stored data. Reference [13] reports the occupancy of block files was 30% for running KNOPPIX 3.8.2 (when the filesystem was EXT2, the volume was 2 GB, and the block size was 256KB).

We applied "ext2optimizer" [13] to the filesystem for HTTP-FUSE CLOOP to create a profile of the actually accessed physical blocks at boot time. We then repacked those blocks into physical blocks located at the "front" of the file store. The ext2optimizer keeps the structure of ext2 filesystem even when the physical blocks rearranged, so the filesystem works the same as before.

Figures 6 and 7 show the effect of ext2optimizer. Figure 6 is the image in which the normal ext2filesystem is translated to block files of HTTP-FUSE CLOOP with accessed data blocks scattered throughout various block files. This requires download of extra data (most 4K blocks are not used in the boot process) and thus increases boot time. Figure 7 shows the effect of ext2optimizer. The accessed data blocks are repacked to increase the efficiency of downloads and reduce boot time.

From the filesystem point of view, ext2optimizer causes fragmentation but reduces download of block files and makes for a quick boot.

Implementation of OS Circular

The guest OS images are distributed by Trusted HTTP-FUSE CLOOP. VMKNOPPIX boots the host OS on a client PC and runs a virtual machine. In the current implementation, Debian GNU/Linux and FreeBSD are bootable on QEMU, KQEMU, KVM and Xen-HVM.

Security Updates

Debian GNU/Linux is supported by a strong community and offers a network update scheme [14] called the APT (Advanced Package Tool) which uses a list of repositories and available packages. If a newer package appears in the repository's list, APT downloads the package and hands the process over to "dpkg" (a medium-level package manager for Debian). The package manager checks the integrity of the package, updates the software, and manages security of the total contents.

Figure 8 details the image of an update on OS circular. When the master OS image on a virtual machine is updated by the "apt-get" command, both a new mapping table file and block files are created. These files are uploaded onto HTTP Servers. The Client PC subsequently sees a list of mapping table files and selects one. Older mapping table files represent less updated OS images. When we want to use an older OS image, the old mapping table file is selected.

We monitored upgrades of Debian packages from 07/Dec/2006 to 11/Dec/2006. A total of 854 packages and a 4 GB virtual disk were translated to 14523 block files (1.9 GB) on 07/Dec/2006. The 104 packages were updated with 3420 block files (335 MB) created on 11/Dec/2006. The new block files and mapping table file were added to HTTP servers and the clients used new OS image by rebooting with the new mapping table file.

Download Sites for OS Circular

Figure 9 shows the global location of OS Circular download sites. We dispersed these sites across the globe to prevent intercontinental connections and provide reasonable latency for downloading block files in US, Europe, and North Asia. Most of these sites are commercial Web hosting services with reasonable prices and a high level of maintenance.

OS Circular has a name resolver which tries to supply close mirror servers. The mirror servers are renamed to the unique sub-domain name which is registered on our DNS, so each client uses DNS lookup at our central site. This DNS is operated by DNS-Balance [15] which resolves the sub-domain name as the nearest mirror server (by using routing information offered by RADB.net, the Routing ASSET Database). It is a server-side solution instead of "netselect" on the client. Figure 10 shows an example of DNS-Balance. The degree of accuracy depends on RADB.net, but intercontinental download is almost always avoided.

DNS-Balance is also used for load balancing and fault-tolerance because it can replace a server at the timeout of keep alive of HTTP1.1.

Performance

We measured the boot time of OS Circular on an IBM/Lenovo Think PAD T60 with a 2 GHz Core2 Duo T7200 and 2 GB Memory. The network latency

was synthesized with a “dumynet” to monitor its effect. The synthesized network delay was tested at both 0 and 30 msec to emulate LAN and ISP environments. We also checked the effect of ext2optimizer and DLAHEAD, both of which aimed to reduce total traffic and reduce the effect of network latency.

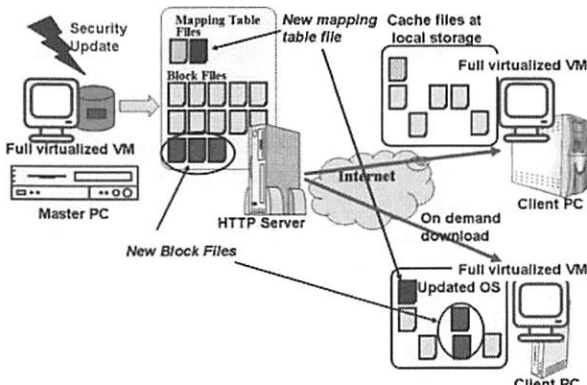


Figure 8: Security update of Trusted HTTP-FUSE CLOOP on OS Circular.

Release	Pkgs	Block Files	Size
Orig. 07Dec2006	884	14,523	1.9 GB
Update 11Dec2006	104	3,430	335 MB

Table 1: The difference of updated blocks.



Figure 9: Download sites for OS circular.

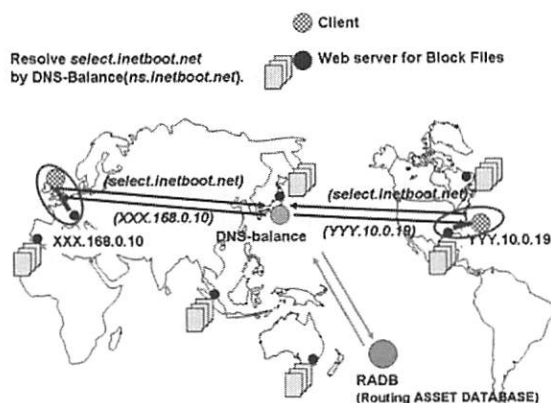


Figure 10: Searching the nearest download site by DNS-Balance.

We compared the boot time on Xen-HVM and KQEMU. Xen-HVM requires the CPU virtualization extension but KQEMU runs on any normal IA32 CPU. The Debian Etch on OS Circular was measured until the gdm (GNOME Display Manager) appeared.

Figures 11 and 12 show the results on Xen-HVM and KQEMU. The three lines show the case of no optimization, ext2optimizer, and ext2optimizer coupled with DLAHEAD. Each line's statistics commenced when GRUB read the kernel and initrd. “Boot” ends when the gdm appears.

Xen-HVM vs. KQEMU

The differences in boot times were not so big on Xen-HVM and KQEMU. In general, Xen-HVM is faster than KQEMU, perhaps twice as fast when the latency is 0 (with no optimization). This advantage was reduced, though, when both ext2optimizer and DLAHEAD were applied. For 30 msec latency, the difference was small and the boot time mostly depended on these optimizations. The results showed the importance of the access speed of virtual disk.

Magnitude of Traffic

The original block device was a 3 GB ext3 formatted disk and included 2 GB of Debian packages. The results show that the original filesystem contains blocks scattered through the filesystem. The ext2optimizer solved this problem.

Without optimization, downloaded disk (network) traffic measured 68 MB on Xen-HVM and 58 MB on KQEMU (the difference was caused by the differing CPU architectures, Core2 Duo of Xen-HVM and Pentium-II of KQEMU) and the BIOS. The BIOS of Xen-HVM is not a fully supported function of QEMU.

We created a disk access profile for Xen-HVM, applied ext2optimizer, and recreated the block files. This reduced the downloaded block files for Xen-HVM to 40% of its original (from 68MB to 27MB) HVM and for KQEMU to 47% of its original (from 58MB to 27MB).

DLAHEAD downloads soon-to-be-accessed block files with four parallel downloading connections. DLAHEAD usually downloads block files before they are requested but sometimes it can't keep up. At that time, some block files are downloaded twice, increasing the traffic. Figures 11 and 12 showed these effects were small, less than 30 MB. The network latency reduction techniques were effective and enabled a quick boot.

Boot Time

Tables 2 and 3 show the boot times for Xen-HVM and KQEMU. To compare them, we added the boot time of the USB Memory (normal), which is normally installed Debian on the USB Memory, and the boot time of the cached block files on the USB memory (cache).

The difference between normal and cached booting is the effect of compressed block files. In Tables 2

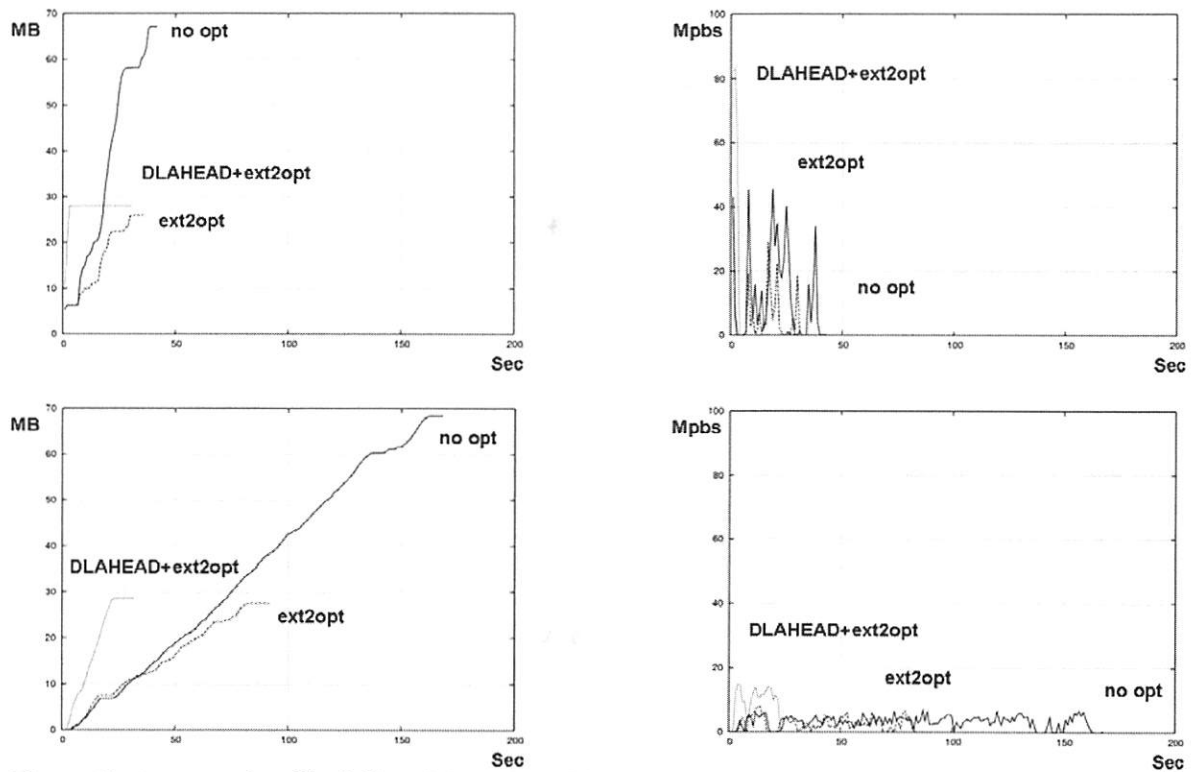


Figure 11: Amount of traffic (left) and throughput (right) at boot time on Xen-HVM under no latency (upper) and 30 msec latency (lower).

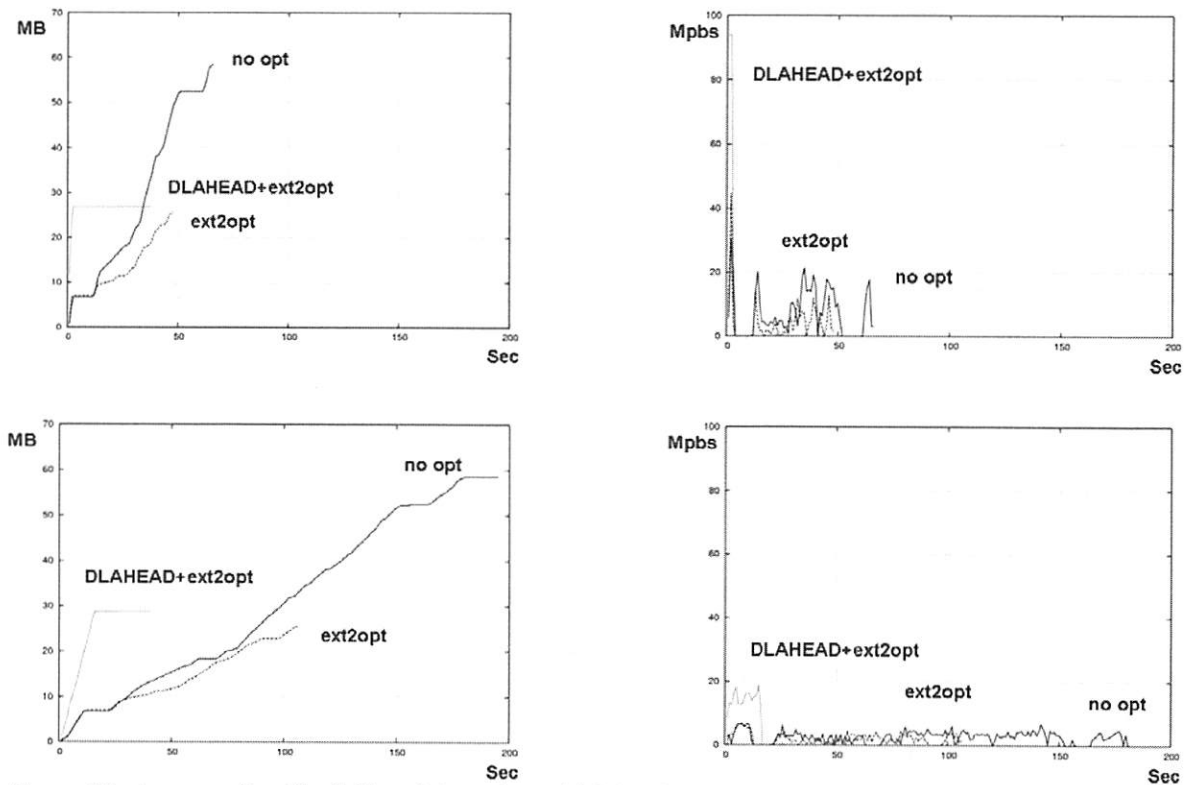


Figure 12: Amount of traffic (left) and throughput (right) at boot time on KQEMU under no latency (upper) and 30 msec latency (lower).

and 3, the difference didn't appear because most time was consumed by boot sequence itself.

Adding the optimizations improved ext2optimizer's boot times in all cases due to the reduction in downloaded block files. This effect grew as network latency was increased. The ratio of boot time with no optimization to that with ext2optimizer approaches the reduction of traffic (to 40% and 47% on Xen-HVM and KQEMU).

When block files were downloaded from the network, DLAHEAD became available. DLAHEAD starts even before a guest OS boots (usually by about five seconds), as soon as the virtual disk is set up for a virtual machine.

If DLAHEAD is not available, the tables show HTTP-FUSE CLOOP's performance is quite vulnerable to network latency. The boot times with 30 msec delay were more than 2x slower than 0 msec latency. The combination of ext2optimizer and DLAHEAD reduced the boot times to less than 41 seconds in each case – close to the case of cached optimized block files. These two types of optimization are necessary for reasonable performance of HTTP-FUSE CLOOP.

Throughput

Since download requests are usually sequential and the size of each download is small (average 100KB), network latency can dramatically reduce a system's performance.

DLAHEAD increased the network throughput to about 90 Mbps on both of Xen-HVM and QEMU when the network latency was 0 msec, finishing the download in five seconds. With 0 msec latency, though, its improvement was diminished. When the network latency was 30 msec, throughput was only 17 Mbps. However, the download finished in 25 seconds and the boot times were almost same as the cases of 0 msec network latency. These results showed the power of DLAHEAD for Internet-based servers.

Discussion

Linking Vulnerability Databases

OS Circular offers a framework for Internet clients. The disk image is updated by the package

manager (though there is no way to authenticate it). The disk image should be linked to Vulnerability Databases to check the contents.

The target vulnerability database is CVE (Common Vulnerabilities and Exposures) [16], operated by the MITRE corporation which includes vulnerability information for packages in each OS. The OVAL (Open Vulnerability Assessment Language) provides a uniform mechanism to report on and control security.

We will apply CVE and offer the information for anonymous users to aid OS selection decisions.

Trusted Boot to Detect Rootkit

The current implementation has to trust VMKNOPPIX. If VMKNOPPIX contains a rootkit or malware on the virtual machine, we have no way to detect it [17, 18].

The Trusted Computing Group (TCG) [19] promotes open standards for hardware-enabled trusted computing. It has released specifications for the Trusted Platform Module (TPM) chip which is often available on current PCs and is used for trusted boot.

We have a plan to integrate trusted boot (e.g., Trusted GRUB [20] and IMA Linux kernel [21, 22]) into VMKNOPPIX. Trusted GRUB and IMA (Integrity measurement Architecture) keep a log of equipped devices and opened files in the TPM. The log is sealed by the key of TPM and sent to a remote attestation server to certify the trusted boot.

Live Update

Some researchers have proposed live update of OSes on virtual machines, e.g., Intel vPro and LUCOS [23]. Virtual machines and the Guest OS communicate with each other and enable live update for security. Currently, OS Circular has no function for live update, but we hope to integrate it in the future.

PlayStation 3

OS Circular has been also been applied to other devices which run Linux. We have even applied OS Circular to a game machine "PlayStation 3"; we call it HTTP-FUSE PS3 Linux [24].

	No optimization	ext2optimiser	DLAHEAD+
Normal	30	28	–
Cache	31	28	–
0 msec latency	42	36	31(+5)
30 msec latency	168	92	32(+5)

Table 2: Boot time on Xen-HVM (sec).

	No optimization	ext2optimiser	DLAHEAD+
Normal	55	46	–
Cache	53	44	–
0 msec latency	66	48	38(+5)
30 msec latency	195	106	41(+5)

Table 3: Boot time on KQEMU (sec).

The most important features are the unified device model and preparation of the block device before booting. The device model of PlayStation3 is unified, and the boot loader, "kboot" [25], is stored in a 4 MB built-in Flash memory. kboot can download a kernel and a miniroot via HTTP from the Internet. This "miniroot" includes the driver for HTTP-FUSE CLOOP and the root filesystem is obtained by HTTP-FUSE CLOOP.

Conclusions

We have proposed OS Circular, a client-centric OS migration system. OS Circular utilizes easily obtained infrastructure, including Web hosting and security update service. OS images are maintained by a security management paradigm on the client, checking the validity of data blocks as they arrive. Performance improvements have enabled OS Circular to achieve performance almost as good as local disks. Future plans include the integration of Trusted Boot and Vulnerability Database checking.

Author Biographies

Kuniyasu Suzaki received the B.Eng. degree and the M.Eng. degree in computer engineering from Tokyo University of Agriculture and Technology. He is a Senior Researcher at National Institute of Advanced Industrial Science and Technology, Japan. His current research interests include trusted computing and OS migration. Reach him electronically at k.suzaki@aist.go.jp.

Toshiki Yagi received the B.Sc. degree in Information Science from the University of Tsukuba. He is a Research Staff member of National Institute of Advanced Industrial Science and Technology, Japan. His current research interests include trusted computing and virtualization. Reach him electronically at yagi-toshiki@aist.go.jp.

Kengo Iijima received the B.Sc. degree in Information Science from the University of Tsukuba. He is a Research Staff of National Institute of Advanced Industrial Science and Technology, Japan. His current research interests include OS migration and virtual stackable block device. Reach him electronically at k-ijima@aist.go.jp.

Nguyen Anh Quynh received the B.Sc. degree and the M.Sc. in Information Technology from Vietnam National University and a Ph.D. in Computer Science from Keio University. He is a Post-Doctoral Research Scientist of Advanced Industrial Science and Technology, Japan. His current research interests include virtualization, trusted computing, and security. Reach him electronically at nguyen.anhquynh@aist.go.jp.

Bibliography

- [1] Suzaki, Kuniyasu, Toshiki Yagi, Kengo Iijima, and Nguyen Anh Quynh, "OS-Circular": A Framework of Internet Client with Xen, Xen Summit

2007 Spring, York Town, NY, April, 2007, http://www.xensource.com/files/xensummit_4/XenSummit07Spring-Suzaki.pdf.

- [2] Suzaki, Kuniyasu, Toshiki Yagi, Kengo Iijima, Junichi Tsukamoto, Megumi Nakamura, and Seiji Munetoh, "OS Circulation Environment: Trusted HTTP-FUSE Xenoppix," Linux Conference Australia 2007, Sydney, Australia, January, 2007, http://mirror.linux.org.au/linux.conf.au/2007/video/monday/monday_1450_Virtualisation.pdf.
- [3] Chandra, Ramesh, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam, "The Collective: A Cache Based System Management Architecture," *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, May, 2005.
- [4] Pfaff, Ban, Tal Garfinkel, and Mendel Rosenblum, "Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks," *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, San Jose, CA, May, 2006.
- [5] OS Zoo, <http://www.oszoo.org/>.
- [6] FLOZ, http://www.oszoo.org/wiki/index.php/Free_Live_OS_Zoo.
- [7] Moka5's LivePC, <http://www.moka5.com/products/>.
- [8] Cáceres, Ramón, Casey Carter, Chandra Narayanaswami, and Mandayam Raghunath, "Reincarnating PCs with Portable SoulPads," *Proceedings on the 3rd Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*, Seattle, WA, June, 2005.
- [9] Kozuch, Michael, and Mahadev Satyanarayanan, "Internet Suspend/Resume," *Proceedings on the Fourth IEEE Workshop on 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*, Washington, DC, June, 2002.
- [10] Quinlan, Sean and Sean Dorward, "Venti: A New Approach to Archival Storage," *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, Monterey, CA, January, 2002.
- [11] Stateless Linux, <http://fedoraproject.org/wiki/StatelessLinux>.
- [12] FUSE (File system in USEr-space), <http://fuse.sourceforge.net/>.
- [13] Kitagawa, Kenji, Tan Hideyuki, Daisuke Abe, Daisaku Chiba, Kuniyasu Suzaki, Kengo Iijima, and Toshiki Yagi, "File System (Ext2) Optimization for Compressed Loopback Device," *Linux Kongress 2006*, http://www.linux-kongress.org/2006/abstracts.html#3_5_1.
- [14] Krafft, Martin F., *The Debian System: Concepts and Techniques*, Open Source Press GmbH, 2005.
- [15] Yokota, Hiroshi, Shigetomo Kimura, and Yoshihiko Ebihara, "A Proposal of DNS-Based Adaptive Load Balancing Method for Mirror Server Systems and Its Implementation," *Proceedings*

on the 18th International Conference on Advanced Information Networking and Applications, March, Fukuoka, Japan, 2004.

- [16] CVE, <http://cve.mitre.org/>.
- [17] King, Samuel T., Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch, "SubVirt: Implementing Malware With Virtual Machines," *Proceedings on the IEEE Symposium on Security and Privacy*, Berkeley, CA, May, 2006.
- [18] Garfinkel, Tal, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proceedings on the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, NY, October, 2003.
- [19] Trusted Computing Group, <https://www.trusted-computinggroup.org>.
- [20] Trusted GRUB, <http://trousers.sourceforge.net/grub.html>.
- [21] IMA (Integrity Measurement Architecture), http://domino.research.ibm.com/comm/research_people.nsf/pages/sailer.ima.html.
- [22] Sailer, Reiner, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," *Proceedings on the 13th USENIX Security Symposium*, San Diego, CA, August, 2004.
- [23] Chen, Haibo, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew, "Live Updating Operating Systems Using Virtualization," *Proceedings on the 2nd International Conference on Virtual Execution Environments*, Ottawa, Canada, June, 2006.
- [24] Yagi, Toshiki, and Kuniyasu Suzaki, "HTTP-FUSE PS3 Linux: An Internet Boot Framework With kboot," *CELF Embedded Linux Conference (ELC'07)*, Santa Clara, Ca, April, 2007, <http://www.celinux.org/elc2007/>.
- [25] kboot, <http://kboot.sourceforge.net/>.

Secure Isolation of Untrusted Legacy Applications

Shaya Potter, Jason Nieh, and Matt Selsky – Columbia University

ABSTRACT

Existing applications often contain security holes that are not patched until after the system has already been compromised. Even when software updates are available, applying them often results in system services being unavailable for some time. This can force administrators to leave system services in an insecure state for extended periods. To address these system security issues, we have developed the PeaPod virtualization layer. The PeaPod virtualization layer provides a group of processes and associated users with two virtualization abstractions, pods and peas. A pod provides an isolated virtualized environment that is decoupled from the underlying operating system instance. A pea provides an easy-to-use least privilege model for fine grain isolation amongst application components that need to interact with one another. As a result, the system easily enables the creation of lightweight environments for privileged program execution that can help with intrusion prevention and containment. Our measurements on real world desktop and server applications demonstrate that the PeaPod virtualization layer imposes little overhead and enables secure isolation of untrusted applications.

Introduction

Security problems can wreak havoc on an organization's computing infrastructure. To prevent this, software vendors frequently release patches that can be applied to address security issues that have been discovered. However, software patches need to be applied to be effective. It is not uncommon for systems to continue running unpatched applications long after a security exploit has become well-known [25]. This is especially true of the growing number of server appliances intended for very low-maintenance operation by less-skilled users. Furthermore, by reverse engineering security patches, attackers have been able to release exploits less than a month after the vulnerability is patched [16].

This impacts system administrators, as even with security patches being released, one cannot always apply them in a timely manner. First, many security patches require that the system service being patched be taken off-line, thereby making it unavailable. Patching an operating system can result in the entire system having to be down for some period of time. If a system administrator chooses to fix an operating system security problem immediately, he risks upsetting his users because of loss of data. Therefore, a system administrator must schedule downtime in advance and in cooperation with all the users, leaving the computer vulnerable until repaired. Furthermore, just because a security patch is released, does not mean it will apply successfully to one's system. If the system service is patched successfully, the system downtime may be limited to just a few minutes during the reboot. However, if the patch is not successful, downtime can extend for many hours while the problem is diagnosed and a solution is

found. Therefore, a system administrator will have to delay applying the security patch until one is sure that it will cause only a minimum amount of downtime.

Second, many system services in use today are supplied as *appliances*. Just like one's physical appliances are simple single task machines, computing appliances, be they commercial appliances, such as a TiVo or a NetApp Filer, or a simplified appliance a corporation deploys internally, such as a web or mail server appliance, are simplified single task systems. A primary advantage of computing appliances is that they can be deployed very easily by less-skilled users. However, this can result in them being set up, left running, and forgotten about since they "just work." As with all software systems, they will suffer from bugs, some of which can have large security implications. Since these appliances are meant to be put into use by people who are not skilled in system administration, one can end up deploying a large number of systems that are vulnerable to be taken over and used maliciously, without the owner of the appliance having any knowledge that this has occurred. Today, actively used personal machines are being actively taken over and used as part of large bot-nets without any knowledge of the owners of the machines [6]. In the future where large numbers of computing appliances will be deployed, this problem will become significantly worse.

There are many principles that are used to increase the security of a software system and limit the damage that can occur if security is breached [26]. One of the most important is ensuring that one operates in a *Least Privilege* environment. Least Privilege environments requires that a user or a program only have access to the resources that are required to complete their

job. Even if the user or service's environment is exploited, the attacker will be constrained. For a system with many distinct users and uses, designing a least privilege system can prove to be very difficult as many independent application systems can be used in many different and unknown ways. On the other hand, securing a single service, such as a software appliance, is more tractable due to the limited nature of what the service accesses.

A common approach to providing least privilege environment to a single service is a sandbox container environment. Many sandbox container environments have been developed to isolate untrusted applications, however, many of these approaches have suffered from being too complex and too difficult to configure to use in practice, and have often been limited by an inability to work seamlessly with existing system tools and applications. Virtual machine monitors (VMMs) offer a more attractive approach by providing a much easier-to-use isolation model of virtual machines, which look like separate and independent systems apart from the underlying host system. However, because VMMs need to run an entire operating system instance in each virtual machine, the granularity of isolation is very coarse, enabling malicious code in a virtual machine to make use of the entire set of operating system resources. Multiple operating instances also need to be maintained, adding administrative overhead.

A primary problem with a sandbox container that attempts to isolate a single service is that many services are composed of many interdependent programs. Each individual application that makes up the service has their own set of requirements. However, since they will all be run within the same sandbox container, each individual application will end up with access to the superset of resources that are needed by all the programs that make up the service, thereby negating the least privilege principle. One cannot divide the programs into distinct sandbox container environments since many programs are interdependent and expect to work from within a single context.

We present PeaPod, a virtualization layer that provides an easy-to-use abstraction that can be used at the granularity of individual applications. The PeaPod virtualization layer provides virtual machine isolation without the need to run multiple operating system instances. PeaPod further enables fine-grain isolation among application components that may need to interact within a single machine environment. PeaPod provides its functionality without modifying, recompiling, or relinking applications or operating system kernels.

PeaPod combines two key virtualization abstractions in its virtualization layer. First, it leverages the pod (Process Domain) [20, 22] to provide a sandbox container for entire services to run within. A pod is a lightweight environment that mirrors the underlying operating system environment. PeaPod isolates processes

in pods from underlying system by associating virtual identifiers with operating system resources and only allowing access to resources that are made available within the pod virtualized namespace. Since the pod virtualization layer provides a virtual machine like environment, it also defines its own set of users, which can be distinct from those supported by the underlying system. Since it does not run an operating system instance, a pod prevents malicious code from making use of an entire set of operating system resources. Second, it introduces peas (Protection and Encapsulation Abstraction). A pea is an easy-to-use least privilege mechanism that enables further isolation among application components that need to share limited system resources within a pod. It can prevent compromised application components from attacking other components within the same pod. A pea provides a simple resource-based model that restricts access to other processes, IPC, file system, and network resources within a pod.

PeaPod improves upon previous approaches by not requiring any operating system modifications, as well as avoiding the *time of check, time of use* race conditions that affect many of them [31]. For instance, unlike other approaches that perform file system security checks at the system call level and therefore do not check the actual file system object that the operating system uses, PeaPod leverages stackable file system to integrate directly into the kernel's file system security framework. PeaPod is designed to avoid the time of check, time of use race conditions that affect previous approaches by performing all file system security checks within the regular file system security paths and on the same file system objects that the kernel itself uses.

This paper describes how the PeaPod system can isolate applications to limit their ability to attack a system. The next section describes the PeaPod's virtualization abstractions in further detail followed by the virtualization architecture to support PeaPod. The next two sections provide a security analysis of the PeaPod system as well as examples of how to use PeaPod. Then the experimental results evaluating the overhead associated with PeaPod and measures the system performance of providing secure isolation for several application scenarios are presented followed by related work. Finally, we present some concluding remarks.

PeaPod Model

The PeaPod model is based on a virtualization abstraction called a pod (Process Domain). A pod looks just like a regular machine and provides the same application interface as the underlying operating system. Pods can be used to run any application, privileged or otherwise, without modifying, recompiling, or relinking applications. This is essential for both easy-of-use and protection of the underlying system, since applications not executing in a pod offer an opportunity to attack the system. Processes within a pod

can make use of all available operating system services, just like processes executing in a traditional operating system environment.

A pod does not run an operating system instance, it instead provides a virtualized machine environment by providing a host-independent virtualized view of the underlying host operating system. This is done by providing each pod with its own private, virtual namespace. All operating system resources are only accessible to processes within a pod through the pod's private, virtual namespace.

A pod namespace is private in that only processes within the pod can see the namespace. It is private in that it masks out resources that are not contained within the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory or signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines.

A pod namespace is virtual in that all operating system resources including processes, user information, files, and devices are accessed through virtual identifiers within a pod. These virtual identifiers are distinct from host-dependent resource identifiers used by the operating system. The pod virtual namespace provides a host-independent view of the system by using virtual identifiers that remain consistent throughout the life of a process in the pod, regardless of whether the pod moves from one system to another.

The pod private, virtual namespace enables secure isolation of applications by providing complete mediation to operating system resources. Pods can restrict what operating system resources are accessible within a pod by simply not providing identifiers to such resources within its namespace. A pod only needs to provide access to resources that are needed for running those processes within the pod. It does not need to provide access to all resources to support a complete operating system environment. An administrator can configure a pod in the same way one configures and installs applications on a regular machine.

For example, if one had a web server that just serves static content, one can easily setup a web server pod to only contain the files the web server needs to run and the content it wants to serve. The web server pod could have its own IP address, decoupling its network presence from the underlying system. It could also limit network access to client-initiated connections. If the web server application gets compromised, the pod limits the ability of an attacker to further harm the system since the only resources he has access to are the ones explicitly needed by the service. Furthermore,

there is no need to carefully disable other network services commonly enabled by the operating system that might be compromised within the pod since there is no operating system running in the pod.

Pods can be used in conjunction with peas (Protection and Encapsulation Abstraction). While pods separate processes into separate machine environments, a pea can be used in a pod to provide fine-grain isolation among application components that may need to interact within a single machine environment, such as using interprocess communication mechanisms, including signals, shared memory, IPC messages and semaphores, and process forking and execution.

A pea is an abstraction that can contain a group of processes and restrict those processes in interacting with processes outside of the pea, and limit their access to only a subset of system resources. Unlike a pod, which achieves isolation by controlling what resources are located within the namespace, a pea achieves isolation levels by controlling what system resources within a namespace its processes are allowed to access and interact with. For example, a process in a pea can see file system resources and processes available to other peas within a single pod, but can be restricted from accessing them. Unlike processes in separate pods, processes in separate peas in a single pod share the same namespace and can be allowed to interact using traditional interprocess communication mechanisms. Processes can also be allowed to move from one pea to another in the same pod. However, by default processes in separate peas cannot access any resource that is not made available to its pea, be it a process pid, IPC key or file system entry.

Peas can support a wide range of resource restriction policies. By default, processes contained in a pea can only interact with other processes in the same pea. They have no access to other resources, such as file system and network resources or processes outside of the pea. This provides a set of fail safe defaults, as any extra access has to be explicitly allowed by the administrator.

The pea abstraction allows for processes running on the same system to have varying levels of isolation, by running in separate peas. Many peas can be used side by side to provide flexibility in implementing a least privilege system for programs that are composed of multiple components that must work together, but do not all need the same level of privilege. One usage scenario would be to have a severely resource limited pea in which a privileged process executes but allows the process to use traditional UNIX semantics to work with less privileged programs that are in less resource restricted peas.

For example, peas can be used to allow a web server appliance the ability to serve dynamic content via CGI in a more secure manner. Since the web

server and the CGI scripts need separate levels of privilege, as well as different resource requirements, they should not have to run within the same security context. By configuring two separate peas for a web service, one for the web server to run within, and a separate for the specific CGI programs it wants to execute, one limits the damage that can occur if a fault is discovered within the web server. If one manages to execute malicious code within the context of the web server, one can only make use of resources that are allocated to the web server's pea, as well as only execute the specific programs that are needed as CGIs. Since the CGI programs will also only run within their specific security context, the ability for malicious code to do harm is severely limited.

Peas and pods together provide secure isolation based on flexible resource restriction for programs as opposed to restricting access based on users. Peas and pods also do not subvert underlying system restrictions based on user permissions, but instead complement such models by offering additional resource control based on the environment in which a program is executed. Instead of allowing programs with root privileges to do anything they want to a system, PeaPod enables a system to control the execution of such programs to limit their ability to harm a system even if they are compromised.

PeaPod Virtualization

To support the PeaPod virtualization abstraction design of secure and isolated namespaces on commodity operating systems, we employ a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This thin virtualization layer is used to translate between the PeaPod namespaces and the underlying host operating system namespace. It protects the host operating system from dangerous privileged operations that might be performed by processes within the PeaPod, as well as protecting those processes from processes outside of the PeaPod on the host. It also enables program-based resource restriction for file access, device access, network access, root privileges, process interactions, and process transitions among peas.

Pod Virtualization

Pods are supported using virtualization mechanisms that translate between pod virtual resource identifiers and operating system resource identifiers. Every resource that a process in a pod accesses is through a *virtual name*, which corresponds to an operating system resource identified by a *physical name*. When an operating system resource is created for a process in a pod, such as with process or IPC key creation, instead of returning the corresponding physical name to the process, the pod virtualization layer catches the physical name value, creates a shadow identifier with a

private virtual name that maps to the physical name and returns the private virtual name to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches it, and replaces it with the appropriate physical name. The key pod virtualization mechanisms used are a system call interposition mechanism and the chroot utility with file system stacking to provide each pod with its own file system namespace that can be separate from the regular host file system.

Pods can take advantage of the regular user identifier (UID) security model to support multiple security domains on the same system running on the same operating system kernel. For example, since each pod can have its own private file system, each pod can have its own `/etc/passwd` file that determines its list of users and their corresponding UIDs. Since the pod file system is separate from the host file system, a process running in the pod is effectively running in a separate security domain from another process with the same UID that is running directly on the host system. Although both processes have the same UID, each process is only allowed to access files in its own file system namespace. Similarly, multiple pods can have processes running on the same system with the same UID, but each pod effectively provides a separate security domain since the pod file systems are separate from one another. Since each pod provides a separate security domain, it needs to be viewed as if it is a distinct machine. For instance, if two physical machines share a writable file system, an attacker could leverage flaws in one machine to get programs on the shared file system that can be used to exploit the second one. While there is value in sharing file system data between pods, one has to use the same care in verifying the shared file system data with multiple pods as one would with multiple independent machines.

Because the root UID 0 is privileged and treated specially by the operating system kernel, pod virtualization also treat UID 0 processes inside of a pod in a special way to prevent them from breaking the pod abstraction, accessing resources outside of the pod, and causing harm to the host system. While a pod can be configured for administrative reasons to allow full privileged access to the underlying system, we focus on the case of pods for running application services that do not need to be used in this manner. Pods do not disallow UID 0 processes, which would limit the range of application services that could be run inside pods. Instead, pods provide restrictions on such processes to ensure that they function correctly inside of pods [22].

While a process is running in user space, the UID it runs as does not have any effect. Its UID only matters when it tries to access the underlying kernel via one of the kernel entry points, namely devices and system calls. Since a pod already provides a virtual file system that includes a virtual `/dev` with a limited

set of secure devices, the device entry point is already secured. The only system calls of concern are those that could allow a root process to break the pod abstraction. Only a small number of system calls can be used for this purpose [22].

Pea Virtualization

Peas are supported using virtualization mechanisms that label resources and enforce a simple set of configurable permission rules to impose levels of isolation among process running within a single pod. For example, when a process is created via the `fork()` and `clone()` system calls, its process identifier is tagged with the identifier of the pea in which it was created. Peas leverage the pod's shadow pod process identifier and also place it in the same pea as its parent process. A process's ability to access pod resources is then dictated by the set of access permissions rules associated with its pea. Like pod virtualization, the key pea virtualization mechanisms used are a system call interposition mechanism and file system stacking for file system resources.

Pea virtualization employs system call interposition to wrap existing system calls to enforce restrictions on process interactions by controlling access to process and IPC virtual identifiers. Since each resource is labeled with the pea in which it was created, the system call interposition mechanism checks if the pea labels of the calling process and the resource to be accessed are the same. For example, if a process in one pea would try to send a signal to another process in a separate pea by using the `kill` system call, the system would return an error value of `EPERM`, as the process exists, but this process has no permission to signal it. On the other hand, a parent is able to use the `wait` system call to clean up a terminated child process's state, even if that child process is running within a separate pea since `wait` does not modify a process by affecting its execution. This is analogous to a regular user being able to list the meta data of a file, such as owner and permission bits, even if the user has no permission to read from or write to the file.

When a new process is created, it executes in the pea security domain of its parent. However, when the process executes a new program, one wants the ability to transition the pea security domain the new program is executing within. Therefore, peas support a single type of pea access transition rule that lets a pea determine how a process can transition from its current pea to another. This transition rule is specified by a program filename and pea identifier. A pea is able to have multiple pea access transition rules of this type. The rule specifies that a process should be moved into the pea specified by the pea identifier if it executes the program specified by the given filename. This is useful when it is desirable to have that new program execution occur in an environment with different resource restrictions. For example, an Apache web server

running in a pea may want to execute its CGI child processes in a pea with different restrictions. Pea transitioning is supported by interposing on the `exec` system call and transitioning peas if the process to be executed matches a pea access transition rule for the current pea. Note that pea access transition rules are one-way transitions that do not enable a process to return to its previous pea unless its new pea explicitly provides for such a transition.

System call interposition is also used to control network access for processes inside the pea. Peas provide two networking access rule types, one to allow processes in the pea to make outgoing network connections on a pod's virtual network adapters, the other to allow processes in the pea to bind to specific ports on the adapter to receive incoming connections. Pea network access rules can allow complete access to a pod network adapter, or only allow access on a per port basis. Since any network access occurs through system calls, peas simply check the options of the networking system call, such as `bind` and `connect`, to ensure that it is allowed to perform the specified action.

Pea virtualization employs a set of file system access rules and file systems stacking to provide each pea with its own permission set on top of the pod file system. To provide a least privilege environment, processes should not have access to file system privileges they do not need. For example, while Sendmail has to write to `/var/spool/mqueue`, it only has to read its configuration from `/etc/mail` and should not need to have write permission on its configuration. To implement such a least privilege environment, peas enable files to be tagged with additional permissions that overlay the respective underlying file permissions. File system permissions determine access rights based on the user identity of the process while pea file permission rules determine access rights based on the pea context in which a process is executed. Each pea file permission rule can selectively allow or deny use of the underlying read, write and execute permissions of a file on a per pea basis. The underlying file permission is always enforced, but pea permissions can further restrict whether the underlying permission is allowed to take effect. The final permission is achieved by performing a bitwise AND operation on both the pea and file system permissions. For example, if the pea permission rule allowed for read and execute, the permission set of `r-x` would be triplicated to `r-xr-xr-x` for the three sets of UNIX permissions and the bitwise AND operation would mask out any write permission that the underlying file system allow. This prevents any process in the pea from opening the file to modify it.

Enforcing on disk labeling of every single file, such as supported through access control lists provided by many modern file systems, is too inflexible if a single underlying file system is going to be used for multiple disparate pods and peas. Since each pea in each pod might make use of similar underlying files

but have different permission schemes, storing the permission data on disk is not feasible. Instead, peas support the ability to dynamically label each file within a pod's file system based on two simple path matching permission rules, *path specific permission rules* and *directory default permission rules*. A path specific permission matches an exact path on the file system. For instance, if there is a path specific permission for `/home/user/file`, only that file will be matched with the appropriate permission set. On the other hand, if there is a directory default permission for the directory `/home/user/` any file under that directory in the directory tree can match it, and inherit its permission set.

Given a set of path specific and directory default permissions for a pea, the algorithm for determining what permission matches to what path starts with the complete path and walks up the path to the root directory until it finds a matching permission rule. The algorithm can be described in four simple steps:

1. If the specific path has a *path specific permission*, return that permission set.
2. Otherwise, choose the path's directory as the current directory to test.
3. If the directory being tested has a *directory default permission*, return that permission set.
4. Otherwise set its parent as the current directory to test and go back to step 3.

If there is no *path specific permission*, the closest *directory default permission* to the specified path becomes the permission set for that path. Since, by default, peas give the root directory `/` a *directory default permission* denying all permissions, the default for every file on the system, unless otherwise specified is deny. This ensures the pea's have a fail safe default setup and do not allow access to any files unless specified by the administrator.

The semantics of pea file permission are based on file path name. If a file has more than one path name, such as via a hard link, both have to be protected by the same permission, otherwise depending on what order the underlying file is accessed the permission set it gets will be determined simply based on the path name that was accessed initially. This issue only occurs on creating the initial set of pea file access permissions. Once the pea is setup, any hard links that are created will obey the regular file system permissions. For instance, one is not allowed to create a hard link to a file that one does not have permission to. On the other hand, if one has permission to access the file, a *path specific permission* rule will be created for the newly created file that corresponds to the permission of the path name it was linked to.

The pea architecture makes use of the pod's stackable file system to integrate the pea file system namespace restrictions into the regular kernel permission model, thereby avoiding *time of check, time of use* race conditions. It accomplishes this by stacking

on top of the file system's lookup function, which fills in the respective file's inode structure, and the permission function, which makes use of the stored permission data to make simple permission determinations. A file system's permission function is a standard part of the operating system's security infrastructure, so no kernel changes are necessary.

Pea Configuration Rules

File System

Many system resources in UNIX, including normal files, directories, and system devices, are accessed via files so controlling access to the file system is crucial. Each pea must be restricted to those files used by its component processes. This control is important for security, because processes that work together do not necessarily need the same access rights to files. All file system access is controlled by path specific and directory default rules, which specify a file or directory and an access right, such as read, write, and execute.

The access values for file rules are *read*, *write*, *execute*, similar to standard UNIX permissions. For convenience, we also define *allow* and *deny*, which are aliases for all three of read, write, and execute and cannot be combined with other access values in the same rules. When a path specific or directory default rule gives access to a file, it implicitly gives execute, but not read or write, access to all parent directories of the file, up to the root directory. On the other hand, if a path specific rule denies access to a directory, then access to both the directory and the directory contents, including subdirectories and files, will be denied, even if a separate rule would give access to subdirectories or files due to it being the best match.

```
pod mailserver {
  pea sendmail {
    path /etc/mail/aliases      read
    path /etc/mail/aliases.db  read
  }
  pea newaliases {
    path /etc/mail/aliases      read
    path /etc/mail/aliases.db  read,write
  }
}
```

Rule 1: Example of Read/Write rules.

Consider the case of the Sendmail mail daemon and the *newaliases* command with regard to the system-wide aliases file. Sendmail runs as the root user and needs to be able to read the aliases file in order to know to where it should forward mail or otherwise redirect it. *newaliases* is a symbolic link to *sendmail* and typically also runs as the root user in order to update the aliases file and convert it into the database format used by the Sendmail daemon. In our example, *newaliases* runs in its own pea and is able to read from `/etc/mail/aliases` and read from and write to `/etc/mail/aliases.db`. Meanwhile *sendmail* runs in another pea and is able to read both files, but not write to them. We use two path

specific rules to express these access rules as described in Rule 1.

```
pod music {
  pea play {
    path /dev/dsp    write
  }
  pea rec {
    path /dev/dsp    read
  }
}
```

Rule 2: Protecting a device.

Similar rules can protect a device like `/dev/dsp`. When a user logs into a system locally, via the console, they are typically given control of local devices, such as the physical display and the sound card. Any application that the user runs has access to read from and write to these local devices, even though this privilege is not necessary. For example, we want to restrict playing and recording of sound files to the play and rec applications, which are part of SoX [27]. Rule 2 describe the rules that provide the appropriate access to the device.

The other file system rule is *dir-default*. It uses the same access values as path, but it is used to specify the default access for files below a directory. Any file or sub-directory will inherit the same access flags since access is determined by matching the longest possible path prefix. Unlike path specific rules, directory default rules can deny access to a directory in general, while still allowing access to specific files. Rule 3 describes a pea that denies access to all files in `/bin`, while only allowing access to `/bin/ls`.

```
pod fileLISTER {
  pea onlyLs {
    dir-default /bin    deny
    path /bin/ls        allow
  }
}
```

Rule 3: Directory default rule.

Transition Rules

In the Sendmail/Procmail use case, sendmail forks off and executes a procmail process to deliver the mail to the user's spool. Procmail needs different security settings, so it must transition from a Sendmail pea to a Procmail pea. Rules must be defined that state to which pea a process will transition upon execution. When a process calls the `execve` system call, we examine the file name to be executed and perform a longest prefix match on all the transition rules. For instance, by specifying a directory for a transition, PeaPod will cause a pea transition to occur for any program executed that is located in that directory, unless there's a more specific transition rule available.

Rule 4 creates a pea for Sendmail and Procmail, and specifies that a process should transition when the procmail program is executed.

```
pod mailserver {
  pea sendmail {
    transition /usr/bin/procmail  procmail
  }
  pea procmail {
  }
}
```

Rule 4: Transition rules.

PeaPod does not provide the ability for a process to transition to another pea besides by executing a new program. If it could, a process could open an allowed file in one pea and then transition to another pea where access to that file was not allowed and thus circumvent the security restrictions.

Networking Rules

PeaPod provides two rules that define the network capabilities a pea exposes to the processes running within it. First, peas are able to restrict a process from instantiating an outgoing connection. Second, peas are able to limit what ports a process can bind to and listen for incoming connections. By default, peas do not let processes make any outgoing connections or bind to any port. While a full network firewall is an important part of any security architecture, it is orthogonal to the goals of PeaPod and therefore belongs in its own security layer.

Continuing the simplified Sendmail/Procmail usage case, an administrator would want to easily confine the network presence of processes running within Sendmail/Procmail peas. By allowing sendmail to make outgoing connections, to enable it to send messages, as well as bind to port 25, the standard port for receiving messages, Sendmail can continue to work normally. On the other hand, processes run within the procmail pea, which will be less restricted, are not allowed to bind to any port for this same reason. On the other hand, programs run from within the procmail pea are allowed to initiate outgoing network connections. This allows programs, such as spam filters that require checking network based information, to continue to work.

```
pod mailserver {
  pea sendmail {
    outgoing  allow
    bind      tcp/25
  }
  pea procmail {
    outgoing  allow
  }
}
```

Rule 5: Networking rules.

Shared Namespace Rules

PeaPod provides a single namespace rule for enabling processes to access the pod's virtual private identifiers that do belong to its personal pea. PeaPod enables peas to be configured to only have access to resources tagged with specific pea identifiers or with the special global pea identifier that enables access to

every virtual private resource in the pod. A common usage of this rule is to enable the creation of a global pea with access to all the resources of a pod, for instance to be enable a process to startup and shutdown services run within a resource restricted pea. Rule 6 describes a pod that has a global pea that is able to access every private virtual identifier in the pod, as well as pea that is able to access the virtual identifiers that belong to one of its sibling peas.

```
pod service {
  pea global_access {
    namespace    global
  }
  pea test1 {
    namespace    test2
  }
  pea test {
  }
}
```

Rule 6: Namespace access rules.

Managing Rules

To make it simpler for administrators to create peas in a pod, we allow groups of rules to be saved to a file and included in the main configuration file for a given PeaPod configuration. These groups of rules would typically describe the minimum resources necessary for a single application. Application packagers can include rule group files in their package and administrators can share rule groups with each other.

path /usr/bin/gcc	read,execute
dir-default /usr/lib/gcc-lib	read,execute
path /usr/bin/cpp	read,execute
path /usr/lib/libiberty.a	read
path /usr/bin/ar	read,execute
path /usr/bin/as	read,execute
path /usr/bin/ld	read,execute
path /usr/bin/ranlib	read,execute
path /usr/bin/strip	read,execute

Rule 7: Compiler rules.

A rule group, such as Rule 7 for a compiler, would be stored in a central location. An administrator uses an *include* rule to reference the external file as part of a development PeaPod. Rule 8 contains the tools necessary to build a Linux kernel from source; and permits access to the source code itself and a writable directory for the binaries.

```
pod workstation {
  pea kernel-development {
    include "stdlibs"
    include "compiler"
    include "tar"
    include "bzip2"
    dir-default /usr/local/src/    read
    dir-default /scratch/binaries allow
  }
}
```

Rule 8: Set of multiple rule files.

These management rules demonstrate PeaPod's ability to distinguish the minimal needs of a program

service in order to execute, while enabling an administrator to define a local policy that can restrict what local resources the program service has access to. The knowledge needed to build a set of rules for a program service that provides the minimal needed set of resources to execute is not always readily available to users of security systems. However, this knowledge is available to the authors and distributors of the system. PeaPod's management rules enable the creation and distribution of rule files that define the minimal set of resources needed to execute a program service, while enabling the local administrator to further define the resources restriction policy.

Security Analysis

Saltzer and Schroeder [26] describe several principles for designing and building secure systems. These include:

- *Economy of mechanism*: Simpler and smaller systems are easier to understand and ensure that they do not allow unwanted access.
- *Fail safe defaults*: Systems must choose when to allow access as opposed to choosing when to deny.
- *Complete mediation*: Systems should check every access to protected objects.
- *Least privilege*: A process should only have access to the privileges and resources it needs to do its job.
- *Psychological acceptability*: If users are not willing to accept the requirements that the security system imposes, such as very complex passwords that the users are forced to write down, security is impaired. Similarly, if using the system is too complicated, users will misconfigure it and end up leaving it wide open.
- *Work factor*: Security designs should force an attacker to have to do extra work to break the system. The classic quantifiable example is when one adds a single bit to an encryption key, one doubles the key space an attacker has to search.

PeaPod is designed to satisfy these six principles. PeaPod provides economy of mechanism using a thin virtualization layer based on system call interposition and file system stacking that only adds a modest amount of code to a running system. The largest part of the system is due to the use of a null stackable file system with 7000 lines of C code, but this file system was generated using a simple high-level file system language [33], and only 50 lines of code were added to this well tested file system to implement the PeaPod file system security. Furthermore, PeaPod changes neither applications nor the underlying operating system kernel. The modest amount of code to implement PeaPod makes the system easier to understand. Since the PeaPod security model only provides resources that are explicitly stated, it is relatively easy to understand the security properties of resource access provided by the model.

PeaPod provides fail safe defaults by only providing access to resources that have been explicitly given to peas and pods. If a resource is not created within a pea, or explicitly made available to that pea, no process within that pea will be allowed to access it. While a pea can be configured to enable access to all resources of the pod, this is an explicit action an administrator has to take.

PeaPod provides for complete mediation of all resources available on the host machine by ensuring that all resource accesses occur through the pod's virtual namespace. Unless a file, process, or other operating system resource was explicitly placed in the pod by the administrator or created within the pod, PeaPod's virtualization will not allow a process within a pod to access the resource.

PeaPod's provide a least privilege environment in two ways. First, pods provide a least privilege environment by enabling an administrator to only include the data necessary for each service. PeaPod can provide separate pods for individual services so that separate services are isolated and restricted to the appropriate set of resources. Even if a service is exploited, PeaPod will limit the attacker to the resources the administrator provided for that service. While one can achieve similar isolation by running each individual service on a separate machine, this leads to inefficient use of resources. PeaPod maintains the same least privilege semantic of running individual services on separate machines, while making efficient use of machine resources at hand. For instance, an administrator could run MySQL and Sendmail mail transfer services on a single machine, but within different pods. If the Sendmail pod gets exploited, the pod model ensures that the MySQL pod and its data will remain isolated from the attacker. Furthermore, PeaPod's peas are explicitly designed to enable least privileged environments by restricting programs in an environment that can be easily limited to provide the least amount of access for the encapsulated program to do its job.

PeaPod provides psychological acceptability by leveraging the knowledge and skills system administrators already use to setup system environments. Because pods provide a virtual machine model, administrators can use their existing knowledge and skills to run their services within pods. Furthermore, peas use a simple resource based model that does not require a detailed understanding of any underlying operating system specifics. This differs from other least privilege architectures that force an administrator to learn new principles or complicated configuration languages that require a detailed understanding of operating system principles.

Similar to least privilege, PeaPod increases the work factor that it would take to compromise a system by simply not making available the resources that attackers depend on to harm a system once they have

broken in. For example, since PeaPod can provide selective access to what program are included within their view, it would be very difficult to get a root shell on a system that does not have access to any shell program.

Usage Examples

We briefly describe three examples that help illustrate how the PeaPod virtualization layer can be used to improve computer security and application availability for different application scenarios. The application scenarios are e-mail delivery, web content delivery, and desktop computing. In the following examples we make extensive use of PeaPod's ability to compose rule files in order to simplify the rules. Instead of listing every file and library necessary to execute a program, we isolate them into a separate rule file to place the focus on the actual management of the service the pea is trying to protect.

E-mail Delivery

For e-mail delivery, PeaPod's virtualization layer can isolate different components of e-mail delivery to provide a significantly higher level of security in light of the many attacks on Sendmail vulnerabilities that have occurred. Consider isolating a Sendmail installation that also provides mail delivery and filtering via Procmail. E-mail delivery services are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, this can provide additional resources to services that do not need them, potentially increasing the damage that can be done to the system if attacked.

```
pod mail-delivery {
  pea sendmail {
    include "stdlibs"
    include "sendmail"
    dir-default /etc
    dir-default /var/spool/mqueue read
    dir-default /var/spool/mail allow
    dir-default /var/run allow
    path /usr/bin/procmail read, execute
    transition /usr/bin/procmail procmail
    bind tcp/25
    outgoing allow
  }
  pea procmail {
    dir-default / allow
    outgoing allow
  }
}
```

Rule 9: E-Mail delivery configuration.

As shown in Rule 9, using PeaPod's virtualization layer, both Sendmail and Procmail can execute in the same pod, which isolates e-mail delivery from other services on the system. Furthermore, Sendmail and Procmail can be placed in separate peas, which allows necessary interprocess communication mechanisms between them while improving isolation. This pod is a common example of a privileged service that has child

helper applications. In this case, the Sendmail pea is configured with full network access to receive e-mail, but only with access to files necessary to read its configuration and to send and deliver email. Sendmail would be denied write access to file system areas such as `/usr/bin` to prevent modification to those executables, and would only be allowed to transition a process to the Procmail pea if it is executing Procmail, the only new program its pea allows it to execute. On mail delivery, Sendmail would then exec Procmail, which transitions the process into the Procmail pea. The Procmail pea is configured with a more liberal access permission, namely allowing access to the pod's entire file system, enabling it to run other programs, such as SpamAssassin. While an administrator could configure programs Procmail executes, such as SpamAssassin, to run within their own Peas, this case keeps them all within a single pea to demonstrate how simple a system can be. As a result, the Sendmail/Procmail pod can provide full e-mail delivery service while isolating Sendmail such that even if Sendmail is compromised by an attack, such as a buffer overflow, the attacker would be contained in the Sendmail pea and not even be able to execute processes, such as a root shell, to further compromise the system.

Web Content Delivery

For web content delivery, PeaPod's virtualization layer can isolate different components of web content delivery to provide a significantly higher level of security in light of common web server attacks that may exploit CGI script vulnerabilities. Consider isolating an Apache web server front end, a MySQL database back-end, and CGI scripts that interface between them. While one could run Apache and MySQL in separate pods, since they are providing a single service, it makes sense to run them within a single pod that is isolated from the rest of the system. However, since both Apache and MySQL are within the pod's single namespace, if an exploit is discovered in Apache, it could be used to perform unauthorized modifications to the MySQL database.

To provide greater isolation among different web content delivery components, Rule 10 describes a set of three peas in a pod: one for Apache, a second for MySQL, and a third for the CGI programs. Each pea is configured to contain the minimal set of resources needed by the processes running within the respective pea. The Apache pea includes the apache binary, configuration files and the static HTML content, as well as a transition permission to exec all CGI programs into the CGI pea. The CGI pea contains the relevant CGI programs as well as access to the MySQL daemon's named socket, allowing interprocess communication with the MySQL daemon to perform the relevant SQL queries. The MySQL pea contains the mysql daemon binary, configuration files and the files that make up the relevant databases. Since Apache is the only program exposed to the outside world, it is the

only process that can be directly exploited. However, if an attacker is able to exploit it, the attacker is limited to a pea that is only able to read or write specific Apache files, as well as exec specific CGI programs into a separate pea. Since the only way to access the database is through the CGI programs, the only access to the database an attacker would have is what is allowed by said programs. Consequently, the ability of an attacker to cause serious harm to such a web content delivery system running with PeaPod's virtualization layer is significantly reduced.

```
pod web-delivery {
  pea apache {
    include "stdlibs"
    path /usr/sbin/apache      read,execute
    path /usr/sbin/apachectl   read,execute
    dir-default /var/www       read,execute
    transition /var/www/cgi-bin cgi
    bind                        tcp/80
  }
  pea cgi {
    include "stdlibs"
    include "perl"
    dir-default /var/www/data   allow
    path /tmp/mysql.sock        allow
  }
  pea mysql {
    include "stdlibs"
    path /usr/sbin/mysqld read, execute
    path /tmp/mysql.sock    allow
    dir-default /usr/share/mysql read
    dir-default /var/lib/mysql allow
  }
}
```

Rule 10: Web delivery rules.

Desktop Computing

For desktop computing, PeaPod's virtualization layer enables desktop computing environments to run multiple desktops from different security domains within multiple pods. Peas can also be used within the context of such a desktop computing environment to provide additional isolation. Many application used on a daily basis, such as mp3 players and web browsers, have had security holes. These holes enable attackers to execute malicious code or gain access to the entire local file system [12, 13]. Rule 11 describes a set of PeaPod rules that are used to contain a small set of desktop applications being used by a user with the `/home/spotter` home directory.

To secure an mp3 player, a pea can be created within the desktop computing pod that restricts the mp3 player's ability to make use of files outside of a special mp3 directory. Since most users store their music within its own subtree, this isn't a serious restriction. Most mp3 content should not be trusted, especially if one is streaming mp3s from a remote site. By running the mp3 player within this fully restricted pea, a malicious mp3 cannot compromise the user's desktop session. This mp3 player pea is simply configured with four file system permissions. A path specific permission that provides access to the mp3 player itself is

required to load the application. A directory default permission that provides access to the entire mp3 directory subtree is required to give the process access to the mp3 file library. A directory-default permission to a directory meant to store temporary files so the mp3 player can be used as a helper application. Finally, a path specific permission that provides access to the /dev/dsp audio device is required to allow the process to play audio.

```
pod desktop {
  pea firefox {
    include "firefox"
    dir-default /home/spotter/.mozilla allow
    dir-default /home/spotter/tmp allow
    dir-default /home/spotter/download allow
    transition /usr/bin/mpg123 mpg123
    transition /usr/bin/acroread acroread
  }
  pea mpg123 {
    include "stdlibs"
    path /usr/bin/mpg123 read, execute
    path /dev/dsp write
    dir-default /home/spotter/tmp allow
    dir-default /home/spotter/music allow
  }
  pea acroread {
    include "stdlibs"
    include "acroread"
    dir-default /home/spotter/tmp allow
  }
}
```

Rule 11: Desktop application rules.

To secure a web browser, a pea can be created within a desktop computing pod that restricts the web browser's access to system resources. Consider the Mozilla Firefox web browser as an example. A Firefox pea would need to have all the files Firefox needs to run accessible from within the pea. Mozilla dynamically loads libraries and stores them along with its plugins within the /usr/lib/firefox directory. By providing a directory default permission that provides access to that directory, as well as another directory default permission that provides access to the user's .mozilla directory, the Firefox web browser can run as normal within this special Firefox pea. Users also want the ability to be able to download and save files, as well as launch viewers, such as for postscript or mp3 files, directly from the web browser. This involves a simple reconfiguration of Firefox to change its internal application.tmp_dir variable to be a directory that is writable within the Mozilla pea. By creating such a directory, such as download within the users home directory, and providing a directory default permission allowing access, we enable one to explicitly save files, as well as implicitly save when one wants to execute a helper application. Similarly, just like Mozilla is configured to run helper applications for certain file types, one would have to configure the Mozilla pea to execute those helper applications within their respective peas. As shown, for an mp3 player, configuring such a pea for these process is fairly simple. The only addition one

would have to make is to provide an additional pea transition permission to the Mozilla pea that tells the PeaPod's virtualization layer to transition the process to a separate pea on execution of programs such as the mpg123 mp3 player or the Acrobat Reader PDF viewer.

Experimental Results

We implemented PeaPod's virtualization layer as a loadable kernel module in Linux that requires no changes to the Linux kernel source code or design. We present some experimental results using our Linux prototype to quantify the overhead of using PeaPod on various applications. Experiments were conducted on two IBM Netfinity 4500R machines, each with a 933 Mhz Intel Pentium-III CPU, 512 MB RAM, 9.1 GB SCSI HD and a 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for the PeaPod on the other client system. The client ran Debian stable with a 2.4.21 kernel.

Name	Description
getpid	average getpid runtime
ioctl	average runtime for the FION-READ ioctl
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed
semget-semctl	IPC Semaphore variable is created and removed
fork-exit	process forks and waits for child that calls exit immediately
Apache	Runs Apache 1.3 under load and measures average request time
Make	Linux Kernel 2.4.21 compile with up to 10 processes active at one time
Postmark	Use Postmark Benchmark to simulate Sendmail performance
MySQL	"TPC-W like" interactions benchmark that uses Tomcat 4 and MySQL 4

Table 1: Application benchmarks.

To measure the cost of PeaPod's virtualization layer, we used a range of micro benchmarks and real application workloads and measured their performance on our Linux PeaPod prototype and a vanilla Linux system. Table 1 shows the seven micro-benchmarks and four application benchmarks we used to quantify PeaPod's virtualization overhead. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call micro-benchmarks directly used the TSC register available on Pentium CPUs to record time-stamps at the significant measurement events. Each time-stamp has an average cost

of 58 ns. The files for the benchmarks were stored on the NFS Server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable. Figure 1 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmarks results.

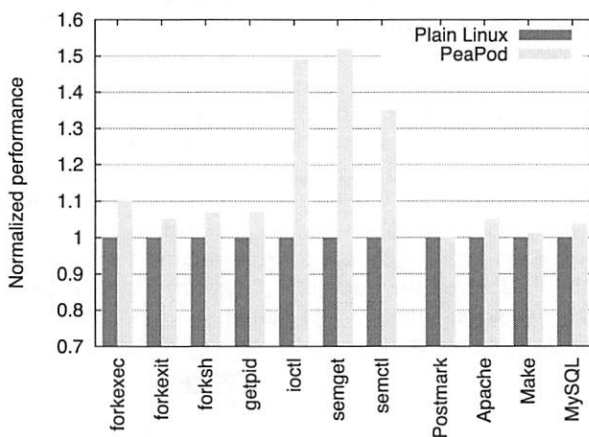


Figure 1: PeaPod virtualization overhead.

The results in Figure 1 show that PeaPod's virtualization overhead is small. PeaPod incurs less than 10% overhead for most of the micro-benchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call `getpid` benchmark is only 7% compared to vanilla Linux, reflecting the fact that PeaPod virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup.

The most expensive benchmarks for PeaPod is `semget+semctl`, which took 51% longer than vanilla Linux. The cost reflects the fact that our untuned PeaPod prototype needs to allocate memory and do a number of namespace translations. The `ioctl` benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. These assignments correspond to saving the four variables that store UID state, assigning them a non privileged UID, and then restoring the original state. This is large compared to the simple `FIONREAD ioctl` that just performs a simple dereference. However, since the `ioctl` is simple, we see that it only adds 200 ns of overhead over any `ioctl`.

For real applications, the most overhead was only four percent, which was for the Apache 1.3 workload, where we used the `http_load` benchmark [21] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web-commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat 4 with a MySQL 4 back-end. The PeaPod overhead for this scenario was less than 2% versus vanilla Linux. These

results are directly comparable to the virtualization results in AutoPod [22] and are effectively the same, demonstrating the additional overhead needed to confine processes into distinct peas is minimal.

Related Work

Many systems have been developed to isolate untrusted applications. NSA's Security Enhanced Linux [19], which is based upon the Flask Architecture [28], implements a policy language that one can use to implement models that enable one to enforce privilege separation. The policy language is very flexible but also very complex to use. The example security policy is over 80 pages long. There is research into creating tools to make policy analysis tractable [2], but the fact that the language is so complex makes it difficult for the average end user to construct an appropriate policy.

System call interception has been used by systems such as Janus [30, 10], Systrace [24], MAPbox [1], Software Wrappers [15], and Ostia [11]. These systems can enable flexible access controls per system call, but they have been limited by the difficulty of creating appropriate policy configurations. TRON [5], SubDomain [7] and Alcatraz [17] also operate at the system call level but focus on limiting access to the underlying file system. TRON allows transitions between different isolation units but requires application modifications to use this feature, while SubDomain supports an implicit transition on execution of a new child process. These systems provide a model somewhat similar to the file system approach used by PeaPod peas. However, peas are designed based on a full-fledged stackable file system that integrates fully with regular kernel security infrastructure and provides much better performance. Similarly, the PeaPod's virtualization layer provide a complete process isolation solution that is not just limited to file system protection.

Safer languages and run-time environments, most notably Java, have been developed to prevent common software errors and isolate applications in language-based virtual machine environments. These solutions require applications to be rewritten or recompiled, often with some loss in performance. Other language-based tools [8, 3] have also been developed to harden applications against common attacks, such as buffer overflow attacks. PeaPod's virtualization layer complements these approaches by providing isolation of legacy applications without modification.

Virtual machine monitors (VMMs) have been used to provide secure isolation [29, 32, 4]. Unlike PeaPod's virtualization layer, VMMs decouple processes from the underlying machine hardware, but tie them to an instance of an operating system. As a result, VMMs provide an entire operating system instance and namespace for each VM and lack the ability to isolate components within an operating system. If a single process in a VM is exploitable, malicious code can make use of it to access and make use of the

entire set of operating system resources. Since PeaPod's virtualization layer decouples processes from the underlying operating system and its resulting namespace, they are natively able to limit the separate processes of a larger system to the appropriate resources needed by them. Furthermore, VMMs require more administrative overhead due to requiring administration of multiple full operating system instances as well imposing higher memory overhead due to the requirements of the underlying operating system.

A number of other approaches have explored the idea of virtualizing the operating system environment to provide application isolation. FreeBSD's Jail mode [14] provides a chroot like environment that processes cannot break out of. However, Jail is limited in what it can do, such as the fact that it doesn't allow IPC within a jail [9], and therefore many real world application will not work. More recently, Linux Vserver [18] and Solaris Zones [23] offer a similar virtual machine abstraction as PeaPod pods, but require substantial in-kernel modifications to support the abstraction. While these system's share the simplicity of the Pod abstraction, they do not provide finer-granularity isolation as provided with peas.

Conclusions

The PeaPod system provides an operating system virtualization layer that enables secure isolation of legacy applications. The virtualization layer supports two key abstractions for encapsulating processes, pods and peas. Pods provide an easy-to-use lightweight virtual machine abstraction that can securely isolate individual applications without the need to run an operating system instance in the pod. Peas provide a fine-grain least privilege mechanism that can further isolate application components within pods. PeaPod's virtualization layer can isolate untrusted applications, preventing them from being used to attack the underlying host system or other applications even if they are compromised.

PeaPod secure isolation functionality is achieved without any changes to applications or operating system kernels. We have implemented PeaPod in a Linux prototype and demonstrated how peas and pods can be used to improve computer security and application availability for a range of applications, including e-mail delivery, web servers and databases, and desktop computing. Our results show that PeaPod's virtualization layer can provide easily configurable and secure environments that can run a wide range of desktop and server Linux applications in least privilege environments with low overhead.

Acknowledgments

This work was supported in part by NSF grants CNS-0426623 and CNS-0717544, a DOE Early Career Award, and an IBM SUR Award.

Author Biographies

Shaya Potter is a Ph.D. candidate in Columbia University's Computer Science department. His research interests are focused around improving computer usage for users and administrators through virtualization and process migration technologies. He received his B.A. from Yeshiva University and his M.S. and M.Phil degrees from Columbia University, all in Computer Science. Reach him electronically at spotter@cs.columbia.edu.

Jason Nieh is an Associate Professor of Computer Science and and Director of the Network Computing Laboratory at Columbia University. He received his B.S. from MIT and his M.S. and Ph.D. from Stanford University, all in Electrical Engineering.

Matt Selsky earned his BS in Computer Science from Columbia University. He has been working at Columbia University since 1999, most recently as an engineer in the UNIX Systems Group. He works on e-mail-related services and is currently pursuing an MS in Computer Science from Columbia University. Reach him electronically at selsky@columbia.edu.

Bibliography

- [1] Acharya, A. and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Applications," *Proceedings of the 9th USENIX Security Symposium*, Aug., 2000.
- [2] Archer, M., E. Leonard, and M. Pradella, "Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux," Technical Report NRL/MR/5540-02-8629, NRL, 2002.
- [3] Baratloo, A., N. Singh, and T. Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.
- [4] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct., 2003.
- [5] Berman, A., V. Bourassa, and E. Selberg, "TRON: Process-specific File Protection for the UNIX Operating System," *Proceedings of the 1995 USENIX Winter Technical Conference*, Jan., 1995.
- [6] CNN Technology, "Expert: Botnets No. 1 Emerging Internet Threat," Jan, 2006, <http://edition.cnn.com/2006/TECH/internet/01/31/furst/index.html>.
- [7] Cowan, C., S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious Server Security," *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, LA, Dec., 2000.
- [8] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,"

- Proceedings of the 7th USENIX Security Conference*, San Antonio, Texas, Jan., 1998.
- [9] FreeBSD Project, *Developer's Handbook*, http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure-chroot.html.
 - [10] Garfinkel, T., "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," *Proceedings of the 2003 Network and Distributed Systems Security Symposium*, Feb., 2003.
 - [11] Garfinkel, T., B. Pfaff, and M. Rosenblum, "Ostia: A Delegating architecture for Secure System Call Interposition," *Proceedings of the 2004 Network and Distributed Systems Security Symposium*, Feb., 2004.
 - [12] GOBBLES Security, *Local/Remote mpg123 Exploit*, http://www.opennet.ru/base/exploits/1042565884_668.txt.html.
 - [13] GreyMagic Security Research, *Reading Local Files in Netscape 6 and Mozilla*, <http://sec.greymagic.com/adv/gm001-ns/>.
 - [14] Kamp P.-H., and R. N. M. Watson, "Jails: Confining the Omnipotent Root," *Proceedings of the 2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May, 2000.
 - [15] Ko, C., T. Fraser, L. Badger, and D. Kilpatrick, "Detecting and Countering System Intrusions Using Software Wrappers," *Proceedings of the 9th Usenix Security Symposium*, Aug., 2000.
 - [16] LaMacchia, B., Personal Communication, Jan, 2004.
 - [17] Liang, Z., V. Venkatakrishnan, and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs," *Proceedings of the 19th Annual Computer Security Applications Conference*, Dec., 2003.
 - [18] Linux VServer Project, <http://www.linux-vserving.org/>.
 - [19] Loscocco, P. and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June, 2001.
 - [20] Osman, S., D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec., 2002.
 - [21] Poskanzer, J., http://www.acme.com/software/http_load/.
 - [22] Potter, S. and J. Nieh, "Reducing Downtime Due to System Maintenance and Upgrades," *Proceedings of the 19th Large Installation System Administration Conference*, San Diego, CA, Dec., 2005.
 - [23] Price, D. and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *Proceedings of the 18th Large Installation System Administration Conference*, Nov., 2004.
 - [24] Provos, N., "Improving Host Security with System Call Policies," *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug., 2003.
 - [25] Rescorla, E., "Security Holes... Who Cares?" *Proceedings of the 12th USENIX Security Conference*, Washington, D.C., Aug., 2003.
 - [26] Saltzer, J. H., and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the 4th ACM Symposium on Operating System Principles*, Oct., 1973.
 - [27] SoX – Sound eXchange, <http://sox.sourceforge.net/>.
 - [28] Spencer, R., S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," *Proceedings of the 8th USENIX Security Symposium*, Aug., 1999.
 - [29] VMware, Inc., <http://www.vmware.com>.
 - [30] Wagner, D., *Janus: An Approach for Confinement of Intrusted Applications*, Master's thesis, University of California, Berkeley, 1999.
 - [31] Watson, R. N. M., "Exploiting Concurrency Vulnerabilities in System Call Wrappers," *Proceedings of the First USENIX Workshop on Offensive Technologies*, Boston, MA, Aug., 2007.
 - [32] Whitaker, A., M. Shaw, and S. D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec., 2002.
 - [33] Zadok, E. and J. Nieh, "FiST: A Language for Stackable File Systems," *Proceedings of the 2000 Annual USENIX Technical Conference*, June, 2000.

Policy-Driven Management of Data Sets

Jim Holl, Kostadis Roussos, and Jim Voll – Network Appliance, Inc.

ABSTRACT

Contemporary storage systems separate the management of data from the management of the underlying physical storage media used to store that data. This separation is artificial and increases the overall burden of managing such systems. We propose a new management layer that unifies data and storage management without any loss of control over either the data or the storage.

The new management layer consists of three basic entities: data sets, which describe the data managed by the system, policies that specify rules for the management of data sets, and resource pools that represent storage that can be used to implement the policies for the data sets. Using data sets and policies, data administrators are able to directly manage their data, even though that requires them to indirectly manage the underlying storage infrastructure. The storage administrator retains control over what the data administrator can do to the storage.

This new management layer provides the infrastructure necessary to build mechanisms that automate or simplify many administrative tasks, including the necessary coordination between the data and storage administrators. We describe Protection Manager, the first tool that uses this management layer to present storage management in the context of data management tasks that the data administrator wants to perform, and evaluate the effectiveness of using this management layer as a way to automate backups.

Introduction

Data management, in the form of the management of files, file systems and structured data, has traditionally been a separate discipline from the management of underlying storage infrastructure. Data administrators have historically concerned themselves with the redundancy, performance, persistence and availability of their data. The storage administrator has focused on delivering physical infrastructure that satisfies the data's requirements. Typically, the storage is configured and then, within the constraints of the configured storage, data management takes place. If the storage requirements of the data change, the data must be migrated to different storage or the underlying storage must be reconfigured. Either process is disruptive and requires multiple domain-specific administrators work closely together.

The separation of administrators is a natural outcome of the incompatible and inflexible infrastructures deployed. Consider a small aspect of data management: redundancy. Owners of data need a certain amount of redundancy for two reasons. The first is to tolerate faults in the physical media, such as disk failures or storage array failures. The second is to tolerate human and software errors that occur during the manipulation of the data. The traditional solution has been to use a variety of incompatible storage platforms from storage vendors to provide differing levels of hardware reliability and to use tape for recovering from human and software errors. In effect, the underlying infrastructure acts as a "proxy" for managing the data.

Modern virtualization features, such as space-efficient replication, result in a storage infrastructure that eliminates much of the incompatibility and inflexibility found in traditional storage environments, but data management and storage management remain separate disciplines. For example, an administrator for a storage system manages FlexVol [3] volumes and aggregates for provisioning storage and uses SnapMirror [10] and SnapVault [4] for making copies of the FlexVol volumes. A data administrator, on the other hand, manages files or structured data and thinks in terms of copies of their files or structured data. Mapping data management requirements onto storage management primitives still requires human interactions and complex processes. As a result, even with a flexible and homogenous storage infrastructure, data management is still done as if the infrastructure was inflexible and incompatible.

To address the gap between what a storage administrator manages and what a data administrator wants to manage, we introduce a new data management framework that we believe can become the basis for a new unified storage and data management discipline. The data management framework consists of a set of three new management objects: data sets, resource pools, and policies. It also consists of several infrastructure components: role based access control [5], a conformance engine, and a centralized database that holds the definitions of these objects.

The data set represents a collection of data and all of the replicas of the data. A data set is independent of any particular physical infrastructure that is currently being used to store its data and is the basic entity

that a data administrator manages. A resource pool is a collection of physical storage resources managed by storage administrators. A policy describes the desired behavior of the data. Role-based access control (RBAC) is used to control access to all managed entities including data sets, policies and resource pools. The conformance engine is responsible for monitoring and configuring the storage infrastructure such that a data set is always in conformance with its policy.

Using this machinery, a storage administrator controls how storage is used by defining specific policies and controls the rights to use these policies with RBAC. Once a storage administrator has configured the policies and access control, data administrators can create or administer his or her data sets by assigning policies from the appropriate authorized set. Because changes in policies allow reconfiguration of the storage, and this reconfiguration is done automatically using the conformance engine, a substantial increase in efficiency results when compared with systems in which management and data management are separated.

For example, in the case of redundancy, the storage administrator first configures resource pools with varying kinds of physical redundancy, such as RAID levels or failover storage capabilities. The storage administrator then constructs policies that can be used to create datasets with varying degrees of physical and backup redundancy. When a data administrator requires storage for a data set, he or she selects a policy that provides the appropriate levels of redundancy. The conformance engine will then provision the actual storage required on the appropriate systems and replicate the data. The conformance engine also monitors the storage to handle scenarios in which the data set is no longer in conformance with the selected policy.

The first implementation of this framework, called Protection Manager, addresses the particular problem of data replica management. The details of this implementation are provided in later sections of this paper.

In the rest of this paper we describe the architecture of the data management framework and how the concepts of data sets, policies and resource pools combine to solve data protection and provisioning problems. We then describe how the machinery was implemented in the Protection Manager tool. We evaluate the extent to which the concepts and framework empower data administrators and simplify administrative tasks as policies change. Finally, we describe future work and make comparisons to other systems.

Related Work

We found a variety of research and systems that targeted aspects of our data management framework, however, none propose or describe an architecture that unifies storage and data management. In particular, none described how storage administrators could securely delegate storage management to data administrators such

that global policies are enforced and maintained. Although some research alludes to the need for a common language between the storage and data administrator we are the first to propose a useable model.

The notion of separating a logical view of data from a physical view in storage is not a new concept. Work done at IBM by Gelb describes a similar model where one of the primary goals was to isolate physical device characteristics from application awareness in homogenous IBM environments [6]. The IBM work recognized the need for automation as a key to managing more data effectively, but appears to concentrate on initial provisioning operations, rather than automating any reconfigurations of storage due to policy changes.

Keaton, et al. propose a model for automating data dependability that is very similar to our data management framework [8]. However, their focus is in the mechanics of automating policy design, not in how their complete model could in fact be architected. Their goal is to try and create a mechanism that would allow a storage administrator to automatically select the policy given the business goals and the underlying technology. Such a mechanism could be incorporated in our data management framework, automating the manual process of policy design.

The problem of automating policy design in general has been studied [1, 2, 9]. However, none of these papers present a general architecture for how such a policy can be used by administrators or software to implement the policy that is designed. The approaches presented could be incorporated into our framework to simplify the design of policies.

The use of policies and storage pools in backup applications is not new. Kaczmarek describes in TSM the use of storage pools and policies [7]. However, the intent of the policies is not to provide a mechanism to allow data administrators to manage their own data or to manage the underlying storage, but to provide TSM with more information as it makes data management decisions.

Policy-based management systems are also nothing new. Commercial systems such as Brocade StorageX, Opsware, and Symantec all utilize this approach for many management tasks. However, these policies are intended to automate tasks within an administration domain, not as a mechanism to delegate control between domains. Patterson, et al. describe the use of policies to reduce the cost of storage management by compressing or deleting old data in the SRM space [11].

Data Set Concepts

Prior to the introduction of data sets and policies, administrators drew little distinction between data and the physical containers associated with that data. As a result, administrators thought of data in terms of where they were located and how they were configured. As data management requirements changed,

storage administrators had to manually map the data to the appropriate storage container and change the configuration associated with that storage container to match those new data management requirements. Managing the mapping and ensuring that the configuration is correct were time-consuming and prone to human error. As a result, configuration changes were infrequent. The net result is that storage was either over-provisioned or under-provisioned in terms of capacity, performance, or redundancy.

Data sets and resource pools do not replace the existing storage and data management containers; they are the management view of those containers. For example, a "data set" refers to the data stored by a FlexVol volume. When storage is provisioned, a FlexVol volume is actually created on the storage system; however, because their configuration is stored externally to the storage system, a data set can exist for the entire life cycle of the data, whereas a particular storage container might not. The data might move to another volume or begin sharing the volume with data in another data set. We expect that, over time, administrators will tend to refer to data sets instead of the underlying storage containers.

Resource Pools

A resource pool contains storage from which data sets can be provisioned. A resource pool can be constructed from either, disks, volumes, aggregates or entire storage systems. If a resource pool is constructed using a storage system then we implicitly add all of the disks and pre-created aggregates on that storage system, including any additional aggregates or disks that are later added. There might be many storage systems or aggregates from multiple storage systems in a single resource pool. The resource pool definition is stored in the external database.

In addition to storage capacity, a resource pool also contains the attributes and capabilities of the underlying storage systems in the pool. These attributes include the data access protocols supported, the performance of the software and storage controller, the reliability of the controller, and the data management software features that are available. These properties are automatically discovered and recorded when storage or storage systems are added to resource pools.

The conformance engine uses the capacity and attributes of resource pools when determining the best location for data to be provisioned. A single resource pool might contain different tiers of storage representing different performance and cost attributes.

A resource pool serves two purposes. The first is to reduce the total number of distinct objects a storage administrator must manage, and the second is to allow greater opportunities for space and load optimizations.

Policies such as thin provisioning limits and RBAC can be applied to whole resource pools, rather than to individual storage systems or components.

Since a resource pool consists of discrete quantities of storage, the larger the pool, the more opportunities to optimize for space and load balancing exist within that resource pool.

User-Defined Properties

Although many properties of resource pool members are discovered automatically, certain properties might be explicitly defined by administrators. This permits more flexibility and control when matching provisioning requests with available resources. For instance, it might be desirable for administrators to add an explicit property related to geography to a resource pool member. This property then might be specified as part of a provisioning policy and matched against available resources in a resource pool that has been assigned this property.

Data Sets

Data sets represent a collection of data and their replicas. In our current implementation, "data set" refers to the data contained within one or more storage containers, not the storage containers. The storage containers used by a data set might change, over time, due to load- or space-balancing actions or policy changes. These changes should be transparent to the users of the data. A data set is provisioned from existing storage containers or from a resource pool according to policy. The data set definition is stored in the external database.

Data sets also have provisioning and data protection policies. The policies apply to all of the data in the data set.

A data set serves four purposes. The first is to allow data administrators to manage their data without requiring that they understand how the storage is configured or where it is physically located. Once the data set has been defined, the administrators only have to choose the policies that best match their data management requirements.

The second purpose of a data set is to reduce the number of managed objects. A data administrator might have a lot of data that must be monitored and managed as a single unit spread across many distinct storage containers, such as an Oracle databases or Exchange applications. A data set allows both the storage and data administrators to manage and view the data as a single unit.

The third purpose is that a data set provides a convenient handle to all of the replicas, allowing administrators to view or restore versions of their data without requiring knowledge of where those versions are (or were) stored.

The fourth is that the data set provides the mapping between the physical storage and the desired behavior associated with its policies. As new storage capabilities are added to the system, or policies are changed, the data management framework can reconfigure the existing storage containers, or possibly migrate data to

new storage containers, to better satisfy the data set policy requirements.

Policies

A policy describes how a data set should be configured. This configuration specifies both how the data set should be provisioned and how it should be protected. In our framework, we treat these as distinct policies.

A provisioning policy consists of a set of attributes that the data set requires from a particular resource pool. Specific attributes include – but are not limited to – cost, performance, availability, how the data can be accessed, and what to do in out-of-space situations.

A data protection policy is a graph in which the nodes represent how storage must be configured on resource pools and the edges describe how the data must be replicated between resource pools. The nodes have attributes such as the retention periods for Snapshot copies. The edges have attributes such as lag threshold or whether mirroring or full backup replication is required.

Policies are used by storage administrators to describe how storage should be provisioned and protected. These policies are then used any time data sets are provisioned, eliminating configuration errors that result from ad-hoc processes. These policies can also be given access control by the storage administrator, such that not all resources and configurations are available to all data administrators. For instance, some data administrators might not have access to policies that require highly reliable or high-performance storage (because of the expense required to satisfy those policies).

Data administrators are free to select any authorized provisioning and protection policies that meet their desired data behavior without regard to how the storage is configured or located.

In practice, the data administrator might assign provisioning or protection policies to data sets currently stored in containers which are incompatible with the policy.

For example, if a data set includes data which resides on a storage system without a SnapMirror license, a policy specifying a mirror relationship between the primary and secondary node cannot be conformed to without reconfiguration. Similarly, if a data set has a policy attached, data administrators might add members to the primary node of the data set which are incompatible with the policy. In both cases, the conformance engine can detect the conflict and explain to the data administrator why the underlying storage needs to be reconfigured or the data migrated. The administrator can cancel the operation or approve the tasks proposed by the conformance engine to bring the data set into conformance.

Conformance Engine

The conformance engine uses the policy to configure the underlying storage. The conformance engine service ensures that the resources used by the data set conform to the attributes described in the associated policy. Much of the automation associated with our management framework is derived from this construct. The conformance engine first monitors the physical environment and then compares the physical environment to the desired configuration specified by the policies. If there is a deviation from configured policy, the software alerts administrators to the violation and, in some cases, automatically corrects the problem.

The conformance engine uses the management object definitions stored in the database when it checks for policy deviations.

The conformance engine is split into two parts. This first part performs a comparison between the policy associated with the dataset and the physical environment, and then prepares a list of proposed actions to bring the data set into conformance. The second part executes the resulting actions.

By breaking this module into two parts, we allow the software to “dry run” any policy changes or requests in order to observe and confirm the resulting actions. This is an important feature in cases in which policy changes might result in highly disruptive actions, such as reestablishing a baseline replica on different storage, or in cases in which administrators simply want to review any changes before committing to them.

For example, the conformance engine periodically compares the data protection relationships protecting the members of a data set with the policy associated with the data set. For each node in the data set, the conformance engine determines the type of data protection relationship, or relationships, which policy dictates this node should be the source of. Then, for each member of the node, the conformance engine attempts to find a data protection relationship of this type originating with the member and terminating with a member of the destination node for each of the node's outgoing connections. If a relationship is not found, a task is generated to construct one.

The results of the conformance run include the task list and an explanation of what actions the system will perform to bring the data set into conformance with the configured policy. For most actions, these tasks are executed automatically, but some require user confirmation. Over time, the set of tasks administrators are willing to automate without user intervention will grow. It is also possible that “unresolvable” tasks will be generated which require user intervention before they can be executed.

Role Based Access Control (RBAC)

RBAC controls management access to all of the management objects. An RBAC service allows a

security administrator to configure which role can perform which operations on which objects. Whenever any operation is attempted on any management object, the RBAC service is consulted for authorization. The RBAC configuration is maintained in the same database as the data sets, policies and resource pools.

The RBAC system allows storage architects to delegate responsibility to data administrators. RBAC allows a storage administrator to safely allow data administrators to select policies and resource pools for their data sets without relinquishing control over the particular resources used.

Once storage resources are in use by a data set or assigned to a resource pool, data administrators, who are not permitted to manage storage directly, are able to indirectly manage them by performing operations on the data set or resource pool.

Comparing the Traditional and Data Set Views

Consider a data center with users that have home directories that are accessed with NFS and CIFS, Oracle deployments over NFS, a Microsoft Exchange deployment and varying degrees of data protection.

In Figure 2 we show the traditional view. In Figure 3 we show the data set view. The traditional view presents a detailed schematic of how the infrastructure is currently configured. The fact that there are several data sets and what the data protection and provisioning policies associated with those data sets are is obscured. Furthermore, the schematic presents a lot of information that is not always important and also hides the fact that some of the differences are irrelevant. The data set view, on the other hand, makes it very clear which data is using the infrastructure and what the provisioning and data protection policies are.

```
taskList GetConformanceTasks(dataSet ds) {
    taskList tl;
    policy p = ds->getPolicy();
    foreach (node n in p->getNodes()) {
        foreach (connection c in n->getOutgoingConnections()) {
            destNode dn = c->getDestNode();
            if (c->isBackupRelationship()) {
                // backups are performed on qtrees
                foreach (qtree q in n->getQtrees()) {
                    bool conformant = FALSE;
                    foreach (destMember dm = q->getDestMembers()) {
                        if (dn->hasMember(dm)) {
                            conformant = TRUE;
                            break;
                        }
                    }
                    if (!conformant) {
                        // generate a task to create a backup relationship
                        tl->addTask(q, c, dn);
                    }
                }
            }
            if (c->isMirrorRelationship()) {
                // volumes are mirrored
                foreach (volume v in n->getVolumes()) {
                    bool conformant = FALSE;
                    foreach (destMember dm = v->getDestMembers()) {
                        if (dn->hasMember(dm)) {
                            conformant = TRUE;
                            break;
                        }
                    }
                    if (!conformant) {
                        // generate a task to create a mirror relationship
                        tl->addTask(v, c, dn);
                    }
                }
            }
        }
    }
    return tl;
}
```

Figure 1: Algorithm used to compare data protection relationships against a configured data protection policy and generate conformance tasks. If no tasks are returned, the data set is in conformance.

The data set view also aggregates the physical infrastructure. Rather than seeing a schematic layout with specific components, we see that there are three different tiers of storage that can be used for protection and provisioning. The data set view is easier to use when performing daily management. The schematic view is important when you need to actually drill down to and manipulate the physical components directly, such as when the hardware breaks or needs to be changed.

Implementation

Protection Manager

The first system to incorporate the framework is Protection Manager, from Network Appliance. It allows backup administrators and storage architects to

coordinate their efforts to construct end-to-end data protection solutions using a workflow-based graphical user interface. In addition to integrating several Network Appliance data protection technologies, Protection Manager uses higher-level abstractions to leverage administrators' time.

Protection Manager builds on the functionality of DataFabric Manager by extending the client/server architecture. Figure 4 shows how Protection Manager and various components that are used to protect storage relate. The dashed lines represent open network APIs.

Storage architects use Protection Manager to define data protection policies and define resource pools consisting of aggregates available for secondary storage

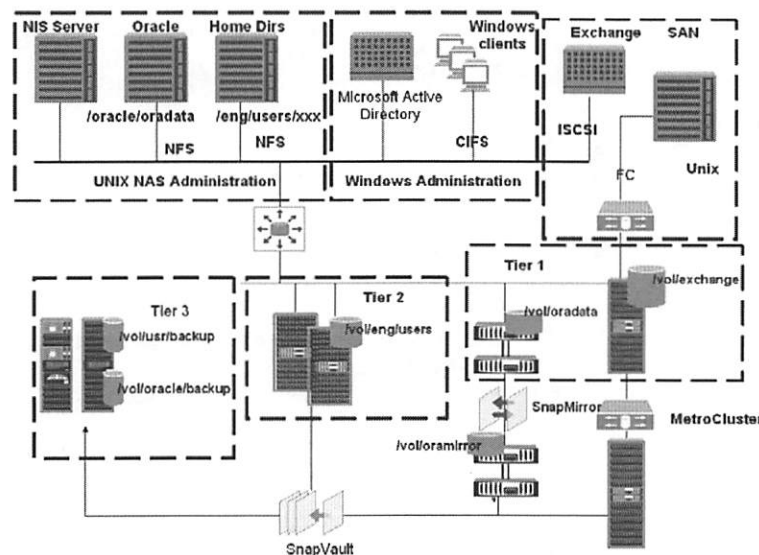


Figure 2: The storage infrastructure view presents a detailed schematic view that obscures how the data uses the infrastructure and omits the data's requirements.

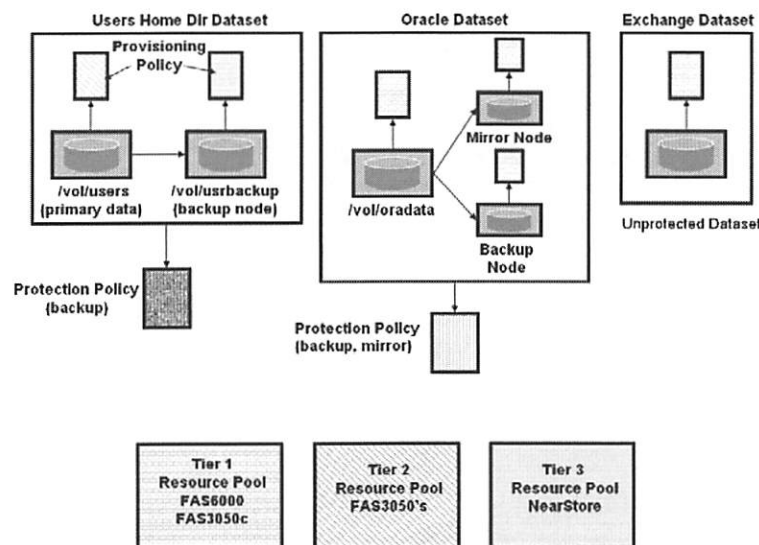


Figure 3: The data set view shows what data is using the infrastructure and what requirements are associated with the data. The data set view abstracts out the details of the underlying storage infrastructure.

provisioning. Then, backup administrators define data sets by adding primary storage objects, such as volumes and qtrees. Based on the data protection policy assigned to the data set (either by the storage architect or by the backup administrator), Protection Manager provisions the required secondary storage from the configured resource pools and creates backup and mirror relationships.

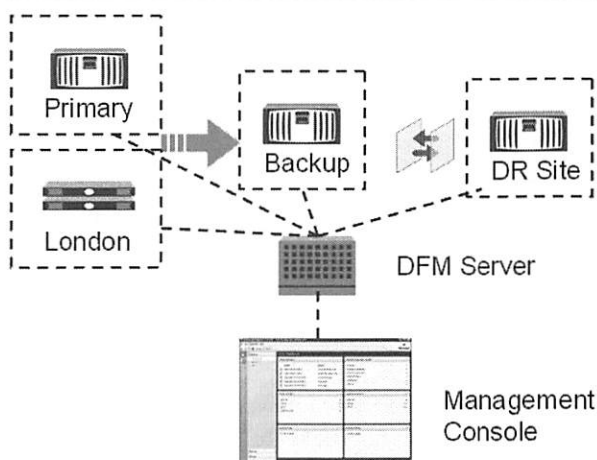


Figure 4: High-level architecture of Protection Manager.

Once the data set is in conformance with its configured data protection policy, Protection Manager continues to monitor its status for deviation and takes action to bring the data set back into conformance. Besides infrastructure errors, out-of-space conditions and policy changes, Protection Manager also monitors changes in the primary data, such as the creation of a new FlexVol volume.

As an example, if the backup administrator creates a data set, "Branch Office," and assigns the policy, "Back up, then mirror," Protection Manager knows that the administrator wants the primary data added to the data set to be protected with a local Snapshot schedule, backed up to secondary storage, and then mirrored to tertiary storage. By assigning resource pools for secondary and tertiary storage provisioning, the administrator tells Protection Manager from where to provision storage. Protection Manager will use the policy's configured schedules for local Snapshot, SnapVault between primary and secondary, and SnapMirror updates between the secondary and tertiary to protect the primary data.

If a new volume is created in a storage system and added to the data set, Protection Manager will discover the new volume and declare that the data set is out of conformance with its configured data protection policy. To rectify the situation, Protection Manager will begin making Snapshot copies of the new volume, provisioning a secondary volume, creating a SnapVault relationship between the new volume and the provisioned secondary volume, provisioning a tertiary

volume, and creating a SnapMirror relationship between the secondary and tertiary volumes. Only then is the data set considered in conformance once again. None of these tasks requires any user interaction with the system.

Geography

Resource pools can be used to group storage resources by performance, availability, owner or any combination thereof, but one of the most interesting uses of resource pools is for geographic grouping. The data administrator knows the geographic location of the primary data in his data set and the configuration of resource pools by the storage administrators gives him visibility into the geographic location of potential backup and mirror sites. This gives him the ability to assign a data protection policy to his data set and select remote storage resources to implement the policy.

Availability of Services

Although the definitions of data sets, resource pools, and policies are maintained in a database within DataFabric Manager in our current implementation, the enforcement mechanism of these policies is configured on storage systems when possible. For instance, the scheduling policy for replication resides in the policy definitions, but the actual scheduling of replication updates should be configured on the associated storage systems. This approach minimizes disruptions in service should a centralized server, such as DataFabric Manager, fail.

The DataFabric Manager server itself can be configured to run in a high availability configuration such as Microsoft Cluster Server.

Impact

In this section, we first evaluate to what extent Protection Manager, using our data management framework, enables division of labor between administrators, empowers data administrators to configure storage and to what extent storage is automatically reconfigured when data protection policy changes. We then explore whether our data management framework, as implemented in Protection Manager, actually simplifies administration.

Division of Labor Between Administrators

Storage administrators and data administrators have different organizational roles. In order to successfully configure data protection, each administrator must coordinate his efforts with the other. Only the data administrator can identify the subset of stored data which constitutes a particular data set and only the storage administrator can identify pools of resources for the data administrator to use.

In a traditional environment, the data administrator must construct a request describing the storage containers he wants protected and either the degree of protection required or enough information about the

use of the data set so that the storage administrator can determine the degree of protection required. This communication could take the form of a help desk ticket or an email. Either way, it must be processed manually and potential for human error is high. Furthermore, the cost of the interaction discourages frequent changes to requirements and encourages “over provisioning” whereby data administrators request more resources than they need to avoid additional requests later.

With the Protection Manager tool, the storage administrator defines resource pools and data protection policies and delegates access to them to the data administrator. Later, the data administrator defines a data set and assigns the appropriate data protection policy. The data administrator only has to know which policy is required for the class of data in his data set. In the definition of the policy, the storage administrator has already encapsulated what that means in terms of schedules, backup and mirror relationships and retention times.

By eliminating the need to coordinate the efforts of two administrators every time a data set is protected, the Protection Manager tool simplifies the data protection process.

Empowering Data Administrators to Configure Storage

In this section we evaluate to what extent the data management framework implemented in Protection Manager actually empowers the data administrator to configure storage to match data protection requirements. To perform the evaluation, we focus on the following task: given a data set, which steps does the data administrator have to execute to configure the data set with local Snapshot copies, remote copies of a subset of the Snapshot copies, and mirrors of the remote copies of the Snapshots copies.

Before we describe what the data administrator does using Protection Manager, consider how the storage is traditionally configured. The primary volume must be configured with a Snapshot schedule. On the secondary system, a volume must be created along with an appropriate SnapVault schedule. Finally, on the disaster recovery storage system, a volume must be created along with a SnapMirror configuration that copies the data from the secondary volume. To perform these tasks without Protection Manager, the data administrator must have administrative access to the storage systems and understand the details of how to correctly configure each element.

Using the Protection Manager UI pictured in Figure 5, the data administrator first selects a policy that has been created by the storage administrator. The data administrator must then select resource pools that have been created by the storage administrator for the various nodes in the policy. At this point the conformance engine will create the secondary and disaster recovery volumes, configure the Snapshot schedule, set up the SnapVault

relationship and configure the SnapMirror relationship using the APIs available on the storage system.

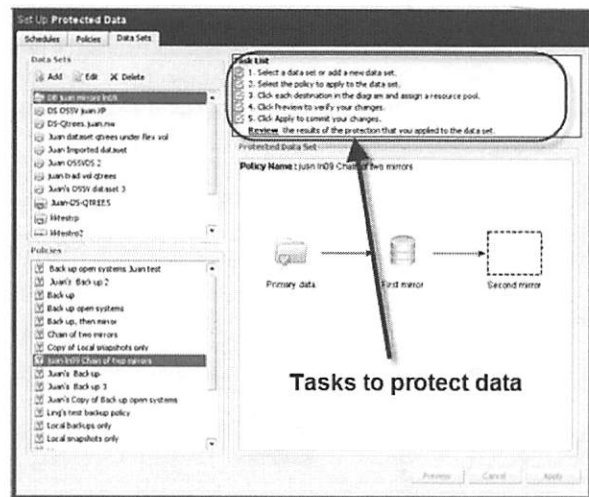


Figure 5: Protection Manager UI.

We have thus shown how, using Protection Manager and our data management framework; a data administrator can configure the underlying storage to meet his or her data protection requirements.

Automatic Reconfiguration of the Storage

In this section we evaluate to what extent Protection Manager allows a storage administrator to reconfigure storage automatically as a result of a change of policy. We assume that the storage administrator has two data protection policies in his or her environment. The first data protection policy only has a local Snapshot schedule. The second data protection schedule has a local Snapshot schedule and a remote backup schedule using SnapVault. To perform the evaluation we focus on the following task: changing the remote backup schedule for the second policy.

Before we describe the steps using the Protection Manager UI, we describe how the storage must be reconfigured. All of the secondary volumes that have SnapVault relationships must have their schedules changed to conform to the new schedule. To perform this task, the storage administrator must log into every secondary and modify each schedule.

Using the Protection Manager UI, the data administrator selects the policy to be modified. The storage administrator then modifies any attributes, for example, how frequently Snapshot copies should be made. The storage administrator then confirms those changes. When the conformance engine runs, it will identify the secondary volumes that must use the new schedule and modify the schedule.

Simplifying Data Management

In this section we evaluate whether the data management framework actually simplifies data management. We consider three metrics. The first is how many management entities must be administered. The

second is how many tasks need to be performed. The third is whether we enable tasks that were impractical before we implemented this new framework.

The data management framework significantly reduces the number of management entities for the data and storage administrator. Rather than having to administer each individual element of a data set (for example, all of the home directories), the data administrator manages a single data set. For the storage administrator, data sets eliminate the need to monitor individual storage containers and relationships to determine whether or not the infrastructure is broken. In addition, the use of resource pools eliminates the need to manually manage space.

The data management framework clearly reduces the number of steps to perform any task. An important consequence of the automation that the conformance engine performs is the elimination of operator errors. This significantly simplifies administration.

The data management framework does enable conformance monitoring, a task that was very difficult if not impractical in large environments. Consider what it would mean to monitor conformance without the framework. Conformance has three elements. The first is a description of the desired behavior. The second is a monitoring of the actual behavior. The third is a comparison of the two. Because the desired behavior is encoded in a way that allows for comparison by software, we are able to perform this task trivially. Without that encoding, a human would have to manually compare the desired behavior to the actual behavior. Furthermore, because a human would be involved, the number of attributes that can be compared would be limited. Protection Manager's conformance engine can check any number of attributes of a data protection policy and can do it more quickly and frequently than a human could.

Conclusions

In this section, we have shown that Protection Manager allows data administrators to configure all aspects of the storage system for data protection. We have shown that Protection Manager can reconfigure the storage if the storage is out of conformance. Finally, we have shown how Protection Manager, using our framework, simplifies data management by reducing the number of managed entities, reducing the number of steps that need to be performed and enabling tasks that were too difficult to be performed manually.

Summary

In the traditional approach to storage and data management, policy is separated from the actual software that implements the policy. As a result, policy has to be carefully translated into operations by storage administrators. Each step described could introduce a human error, requiring significant time and effort to correct. By encoding the policy and being able

to manage both the backup and storage process, using the data set, policy and resource pool abstractions, the storage administrator is able to perform tasks in substantially fewer steps.

Future Work

The task of modeling both cost and performance attributes in policies remains the subject of future work. At this point we have deferred to ad-hoc methods, such as manual classification of systems based on user-defined properties, to guide the provisioning process. Future versions of software will do more automatic classification of resources in this area.

Data set hierarchies are also the subject of future work, especially considering complex application architectures such as SAP or Oracle deployments. These systems place different storage requirements on different subcomponents of each application, which results in different policies. A data set representing the entire application makes sense for performance monitoring or accounting purposes, but the sub-components might be composed of separate data sets, each with unique provisioning and protection policies.

In addition, the migration of data from one storage container to another as a result of space or load balancing has not been explored.

Finally, we intend to explore further refinements of how policies are expressed. The current protection policies are topologies of how data is to be made redundant. We want to explore how administrators can describe the amount of redundancy desired and let the system determine the protection topology.

Conclusions

By introducing the concept of data sets to Network Appliance management software, we have abstracted the management of physical storage containers from the data. As a result, we can now use software to manage the data and adapt to changes in policies or resources automatically, thus making it possible for administrators to manage more data.

Author Biographies

Jim Holl is a graduate of the University of California at Berkeley with a B.S. in Electrical Engineering and Computer Science. He has worked on a variety of software platforms including Solaris and Java at Sun Microsystems and the Universal Agent Platform at Arula Systems. He currently works on storage management software at Network Appliance. Jim can be reached at jimholl@netapp.com.

Kostadis Roussos is a graduate of Brown University with a Sc.B in Computer Science and Stanford University with a MS in Computer Science. He spent the first part of his career writing a thread scheduler at SGI, and then writing a packet scheduler at NetApp. He recently made a major context switch to start

working on simplifying the management of data using storage virtualization. He can be reached at kostadis@netapp.com.

Jim Voll graduated from the University of California at Santa Barbara with a B.S. degree in computer science. He has worked on a variety of system software including compilers, file systems and UNIX operating systems. He has been working at Network Appliance as a Technical Director and is currently focused on storage management software. Jim can be reached at jjv@netapp.com.

Bibliography

- [1] Alvarez, G., E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch and J. Wilkes, "MINERVA: an automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, Vol. 19, Num. 4, pp. 483-518, 2001.
- [2] Anderson, E., S. Spence, R. Swaminathan, M. Kallahalla, Q. Wang, "Quickly Finding Near-Optimal Storage Designs," *ACM Transactions on Computer Systems*, Vol. 23, Num. 4, pp. 337-374, 2005.
- [3] Edwards, J., R. Fair, E. Hamilton, A. Kahn, A. Prakash, E. Zayas, "Flexible Volumes in Data ONTAP," *Network Appliance Technical Journal*, Vol. 2, Num. 2, 2005.
- [4] Fore, A., J. Lyons, K. Trahan, "Beyond Backup: Disk-to-Disk Backup Comes of Age," *WP-7020-0607*, 2007.
- [5] Ferraiolo, D. F. and D. R. Kuhn, "Role-Based Access Control," *15th NIST-NCSC National Computer Security Conference*, pp. 554-563, 1992.
- [6] Gelb, J. P., "System-Managed Storage," *IBM Systems Journal*, Vol. 28, Num. 1, pp. 77-103, 1989.
- [7] Kaczmarski, M., T. Jiang, A. Pease, "Beyond Backup Storage Management," *IBM Systems Journal*, Vol. 42, Num. 2, pp. 322-337, 2003.
- [8] Keeton, K. and J. Wilkes, "Automating Data Dependability," *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, pp. 93-100, Saint-Emilion, France, EW10, ACM Press, New York, NY, July, 2002.
- [9] Keeton, K., C. Santos, D. Beyer, J. Chase, and J. Wilkes, "Designing for Disasters," *USENIX FAST 2004*.
- [10] Patterson, R., S. Manley, M. Federwisch, D. Hitz, S. Kleiman, S. Owara, "SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery," *USENIX FAST*, 2002.
- [11] Zadok, E., J. Osborn, A. Shater, C. Wright, K. Muniswamy-Reddy, J. Nieh, "Reducing Storage Management Costs via Informed User-Based Policies," *IEEE Conference on Mass Storage Systems and Technologies*, April, 2004.

ATLANTIDES: An Architecture for Alert Verification in Network Intrusion Detection Systems

Damiano Bolzoni – University of Twente, The Netherlands

Bruno Crispo – Vrije Universiteit, The Netherlands & University of Trento, Italy

Sandro Etalle – University of Twente, The Netherlands

ABSTRACT

We present an architecture¹ designed for alert verification (i.e., to reduce false positives) in network intrusion-detection systems. Our technique is based on a systematic (and automatic) anomaly-based analysis of the system output, which provides useful context information regarding the network services. The false positives raised by the NIDS analyzing the incoming traffic (which can be either signature- or anomaly-based) are reduced by correlating them with the output anomalies. We designed our architecture for TCP-based network services which have a client/server architecture (such as HTTP). Benchmarks show a substantial reduction of false positives between 50% and 100%.

Introduction

Network intrusion-detection systems (NIDSs) are considered an effective second line of defense against network-based attacks directed to computer systems [4, 11], and – due to the increasing severity and likelihood of such attacks – are employed in almost all large-scale IT infrastructures [1].

The Achilles's heel of NIDSs lies in the large number of *false positives* (i.e., notifications of attacks that turn out to be false) that occur [26]: practitioners [24, 31] as well as researchers [3, 8, 15] observe that it is common for a NIDS to raise thousands of alerts per day, most of which are false alerts. Julisch [16] states that up to 99% of total alerts may not be related to real security issues. Notably, false positives affect both *signature* and *anomaly-based* intrusion-detection systems [2]. A high rate of false alerts is – according to Axelson [3] – the limiting factor for the performance of an intrusion-detection system. False alerts also cause an overload for IT personnel [24], who must verify every single alert, a task that is not only labor intensive but also error prone [9]. Indeed, a high false positive rate can even be *exploited* by attackers to overload IT personnel, thereby lowering the defenses of the IT infrastructure.

The main reason why NIDSs raise false positives is that – quoting Kruegel and Robertson [18] – they are often run without any (or very limited) information about the network resources that they protect (i.e., the context). Chaboya, et al. [6] state that the context knowledge (e.g., network and system configurations)

can improve significantly alert verification. On the other hand, building and updating a database of the configurations or running vulnerability assessment tools (e.g., Nessus [35]) to provide context knowledge is expensive and often not feasible when dealing with complex systems (indeed these activities require additional labor of IT personnel, since the process of using them cannot be completely automated). Most current techniques to improve alert verification are tailored for specific attacks [14, 41] (e.g., worm-like) or support only signature-based NIDSs [33, 36] (e.g., Snort's team has developed a specific plug-in, *flowbits*, to cope with this, but it has limited functionality).

Our thesis is that, in many relevant situations, the context information can be obtained by a systematic (and automatic) *anomaly-based* analysis of the output traffic of the monitored network services; we believe this is possible when the output traffic presents some regularities.

To demonstrate our claims, we have developed *ATLANTIDES* (Architecture for Alert verification in Network Intrusion Detection Systems) an innovative architecture for easing the management of *any* NIDS (be it signature or anomaly-based) by reducing, in an automatic way, the number of false alarms that the NIDS raises. The main idea behind *ATLANTIDES* is simple: a successful attack often causes an *anomaly* in the *output* of the service [44], thus modifying the normal output outcome. Detecting this anomaly can help in reducing false alerts. For instance, a successful SQL Injection attack [43] against a web application often causes the output of SQL table content (e.g., user/admin credentials) rather than the expected web content.

ATLANTIDES, which is completely network-based,² works by analyzing (using n-Gram analysis

²It relies only on information gathered over the network, without involving any host-based component.

¹This research is supported by the research program Sentinels (<http://www.sentinels.nl>). The work of the second author was partially funded by the IST FP6 GridTrust project, contract No. 033827. Part of this work was carried out during the third author's stay at the University of Trento, supported by the GU-IST project Serenity.

[13]) and modeling the normal output payload of the monitored network services that is expected to be sent in response to a client request. This normal output is specific to the site; therefore the derived models reflect – in a way – the network/system context. By correlating the anomalies detected on the output with the alerts raised by the NIDS monitoring the input traffic, we can discard a number of the latter as being false alerts. This way we obtain a system that raises considerably less false positives than the original NIDS, without this correlation system.

Because it is based on output payload analysis, our architecture is designed for TCP-based client/server network services (such as HTTP). Like all (external) payload-based analysis, ATLANTIDES cannot work properly with encrypted data unless the cryptographic keys are provided.

In the past, simple correlations between input and output traffic have already been used to identify possible worm attacks [14, 41]. To the best of our knowledge, ATLANTIDES is the first proposed solution for alert verification that:

- works in combination with both signature-based and anomaly-based NIDSs
- operates in a completely automatic way after a quick setup, without any further human involvement (i.e., reducing the IT personnel overload), thus easing NIDS management

We benchmarked ATLANTIDES in combination with the signature-based NIDS Snort [34, 37], as well as in combination with the anomaly-based NIDS POSEIDON [5]. We carried out benchmarks both on a private data set as well as on the common DARPA 1999 data set [22] (for the sake of completeness and to allow duplication of our results, despite criticism [23, 25]). In seven out of eight cases, our benchmarks show a reduction of false positives between 50% and 100%.

Preliminaries

In this section, we introduce the concepts used in the rest of the paper and explain how false positives arise in signature and anomaly-based systems.

Signature-Based Systems

Signature-based systems (SBSs), e.g., Snort [34, 37], are based on pattern-matching techniques: the NIDS contains a known-attack *signature* database and tries to match these signatures with the analyzed data. When a match is found, an alert is raised. A specific signature must be developed off-line, and then loaded into the database before the system can begin to detect a particular intrusion. One of the disadvantages of SBSs is that they can detect only known attacks: new attacks will be unnoticed till the system is updated, creating a window of opportunity for attackers (and affecting NIDS completeness and accuracy [10, 11]). Although this is considered acceptable for detecting

attacks to, e.g., the OS, it makes them less suitable for protecting web-based services, because of their *ad hoc* and dynamic nature.

False Positives in Signature-Based Systems

SBSs raise an alert every time that traffic matches one of the signatures loaded into the system. Consider for example the *path traversal attack*, which allows to access files, directories, and commands residing outside the (given) web document root directory. The most elementary path traversal attack uses the “../” character sequence to alter the resource location requested in the URL. Variations include valid and invalid Unicode-encoding (“..%u2216” or “..%c0%af”), URL encoded characters (“%2e%2e%2f”), and double URL encoding (“..%255c”) of the backslash character (excerpted from the *WASC Threat Classification* [43]).

To detect these attacks many SBSs (using an out-of-the-box configuration) raise an alert each time they identify the pattern “../” in the incoming traffic. Unfortunately, this pattern could be present in legal traffic too; some Content Management Systems (CMSs) insert relative paths in parameters to load files, which causes SBSs to raise a high number of false alerts. These false alerts can be avoided by deactivating the specific rule. On the other hand, this prevents the NIDS from detecting this sort of attacks.

Tuning Signature-Based Systems

The main reasons why alerts produced by SBSs turn out to be either false or irrelevant include the following:

- Writing signatures for NIDS is a thorny task [32], in which it is difficult to find the right balance between an overly specific signature (which is not able to detect a simple attack variation) and an overly general one (which will classify legitimate traffic as an attack attempt).
- The monitored environment is not susceptible to a certain vulnerability.
- Misconfigured network devices or services producing atypical output (usually, in this case, it is possible to observe recurrent and periodic phenomena).

A good deal of the false positives raised by a SBS can be suppressed by a *tuning* activity: this activity, based on deactivation of unneeded signatures, requires a thorough analysis of the environment by qualified IT personnel. Finally, to remain effective, SBSs require configuration updating to reflect changes in the environment: new vulnerabilities are discovered daily, new signatures are released regularly, and systems may be patched, thereby (possibly adding or) removing vulnerabilities.

Anomaly-Based Systems

Anomaly-based systems (ABSs) use statistical methods to monitor network traffic. Intuitively, an ABS works by training itself to recognize acceptable behavior and then raising an alert for any behavior

outside the boundaries of its training. In the training phase, the ABS builds a model of the normal network traffic. Later, in the operational phase, the ABS flags as an attack any input that significantly deviates from the model. To determine when an input significantly deviates from the model the ABS uses a *distance* function and a *threshold* set by user: when the distance between the input and the model exceeds the threshold, an alarm is raised.

The ABSs' main advantage is that they can detect zero-day attacks: novel attacks can be detected as soon as they take place. Clearly, because of their statistical nature, ABSs are bound to raise a number of false positives, and the value of the threshold actually determines a compromise between the number of false positives and the number of false negatives the IT security personnel is willing to accept.

False Positives in Anomaly-Based Systems

The high false positive rate is generally cited as one of the main disadvantages of anomaly-based systems. The value of the threshold has a direct influence on both false negative and false positive rates [40]: a low threshold (too close to the model) yields a high number of alerts, and therefore a low false negative rate, but a high false positive rate. On the other hand, a high threshold yields a low number of alerts in general (therefore a high number of false negatives, but a low number of false positives). The most commonly used tuning procedure for ABSs is finding an optimal threshold value, i.e., the best compromise between a low number of false negatives and a low (or acceptable) number of false positives. This is typically carried out manually by trained IT personnel: different improving steps may be necessary to obtain a good balance between detection and false positive rates.

Architecture

ATLANTIDES's architecture (see Figure 1) consists of one external and two internal components. The external component is the NIDS monitoring the incoming traffic. We do not make any assumption about it except that it is capable of raising an alert: ATLANTIDES can work together with any kind of NIDS (signature or anomaly-based).

The first internal component is the output anomaly detector (OAD), which is actually an anomaly-based NIDS monitoring the outgoing traffic: the OAD refers to a statistical model describing the normal output of the system, and flags any behaviour that significantly deviates from the norm as the result of a possible attack.

The second internal component is the correlation engine (CE), which tracks (using stateful-inspection [7]) and correlates alerts related to incoming traffic and raised by the input NIDS with the output produced by the OAD.

ATLANTIDES works as follows (see Figure 1). The input NIDS monitors the incoming traffic while,

simultaneously, the OAD (after a training phase) analyses the output of network services. When the input NIDS raises an alert, this is forwarded to the CE, together with the information regarding the communication endpoints (i.e., source and destination IP addresses, source and destination TCP ports as well as sequence numbers and communication status) of the packet that raised the alert. The CE uses a hash-table to store this information, using less than 20 bytes per each entry: thus, the CE does not require much memory to store the information, and ATLANTIDES can handle even a rate of 1000 alerts per second with a total memory space of 1 MB (in case the connections are kept in memory, e.g., for a maximum time of 60 seconds before being dropped). At this time, the alert is not considered an incident yet (it is a *pre-alert*) and is not forwarded immediately to IT specialists.

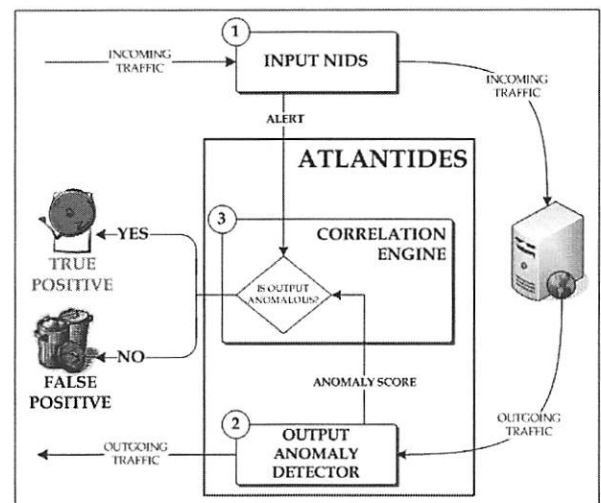


Figure 1: ATLANTIDES's architecture.

Next, the CE marks communication relative to the given endpoints as suspicious and waits for the output of the OAD: if the OAD detects an anomaly in the outgoing traffic related to the tracked communication, then the system considers the alert as an *incident* (i.e., a positive) and the alert is forwarded to the IT specialists for further handling and countermeasure reactions, otherwise it is considered a *false positive* and discarded. The IT personnel can manually set (or adjust) the time value t that the CE waits before dropping an entry from its hash-table, because no output has been produced: during our experiments we fixed this value to 60 seconds. This time could be critical if an attack results in a large data transfer (but in this case the OAD should detect the anomaly in the transferred data) or in the case where attacker is able to delay server response (although this seems quite difficult to realize and the literature does not provide any example of such an attack).

Although a delay is introduced to allow the OAD to process the data sent back to the client, this does not

affect the detection itself: in fact, the delay, in the worst case of no output sent at all, is equal to the time value t . In Appendix A we provide the pseudo-code of our architecture.

It should be clear from the architecture that ATLANTIDES will never raise more false positives than its input NIDS. In fact, the output of the OAD generates false positives or false negatives. The former situation cannot take place because the output of the OAD is evaluated *only* when an alert has already been raised by the input NIDS: the OAD could mistake the alert-related outgoing traffic as anomalous and then forward the alert as a true positive, but this would have happened in any case, if considering the output of the input NIDS only. Thus, the worst case is that a false positive is not suppressed, but any new false alert cannot be generated.

On the other hand, we have to discuss the possibility that ATLANTIDES will introduce additional false negatives (w.r.t. the input NIDS). This happens every time the OAD classifies an alert corresponding to a true attack as a false alert. False negatives are a common problem for alert verification systems (and for ABSs in general). Because of our solution bases its verification on an anomaly-based engine, the threshold used to discern outgoing traffic can be adjusted manually by IT specialists to avoid false negatives (previous proposed solutions cannot be tuned in the same way, e.g., [18]). an effective threshold automatically.

Missing Output Response

What we just described is the most common behavior; nevertheless we have to take into account that there exist attacks which, e.g., aim to disrupt completely the service or that, exploiting a buffer overflow, radically modify the normal execution. In this case, if the OAD does not detect *any* output related to the pre-alert raised by the NIDS, during the time window t , then the pre-alert is considered an incident and is forwarded to an IT security specialist. Although this could be considered rough, because the missing response could occur for different reasons than a successful attack (e.g., an internal error), this strategy does not introduce any additional false negatives/positives, since with a single NIDS (monitoring the incoming traffic) the alert would be forwarded anyway. Furthermore, Chaboya, et al. [6] experimentally verified that most of the buffer overflow attacks against an HTTP server do not produce any output from the attacking requests. Although it is theoretically possible that the attacker crafts a particular payload to send a normal response on the current connection after the exploitation, there exist several difficult technical problems which limit the success of this kind of attack. The attacker must inject an attack payload containing the routines to generate the normal output too (or to jump to the original code where this is done): since exploitable buffers are normally small in size, it could be difficult to include the necessary payload.

Since nowadays attacks against connection-less protocols are less common (see the Common Vulnerabilities and Exposures [39] (CVE) database for detailed statistics), we have designed ATLANTIDES with the explicit goal of reducing false positives when monitoring network services based on the TCP protocol (e.g., HTTP, SMTP and FTP) where a response is typically sent by the server to the client.

Although we do not aim to handle all kinds of possible attacks (e.g., worms or DDoS attacks perpetrated generating a high quantity of legal connections), we believe our solution can improve the accuracy of a NIDS without any additional component installed directly on the monitored hosts (an additional component could affect under certain circumstances host performance, i.e., a high amount of connections).

The OAD

The OAD is basically an anomaly payload-based NIDS, monitoring the output of a network service rather than the input of it. In our embodiment we choose to use the NIDS POSEIDON as the OAD, because we are familiar with it and it gives better results than its leading competitor [5]. POSEIDON is a 2-tier payload-based ABS that combines a neural network with n-gram analysis to detect anomalies. POSEIDON performs a packet-based analysis: every packet is classified by the neural network; then, using the classification information given by the neural network, the real detection phase takes place based on statistical functions considering the byte-frequency distributions (n-gram analysis).

The fact that the OAD is anomaly-based (rather than signature-based) has various advantages. The OAD can adapt to the specific network environment/service, and it does not require the definition of new signatures to detect anomalous output, working in an unsupervised way (after initial setup). Creating and maintaining a set of signatures for outgoing traffic is a thorny and labor-intensive task, as these signatures heavily depend on local applications, and must be updated each time that modifications of the application change its output content. On the other hand, the OAD can simply include these modification in its model, without starting training over. The disadvantage of being anomaly-based is that our OAD needs an extensive (though unsupervised) *training* phase: a significant amount of (normal) traffic data is needed to build an accurate model of the service we monitor.

Setting the Threshold

As we mentioned in Section Anomaly-Based Systems, in ABSs completeness and accuracy are intrinsically related and heavily influenced by the *threshold value*. Here, we call *completeness* the ratio $TP/(TP + FN)$ and *accuracy* the ratio $TP/(TP + FP)$, where TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives raised during the benchmarks. Our experiments show that setting the threshold at $3 \frac{t_{max}}{4}$, usually yields

reasonably good results, where t_{max} is the maximum distance between the analyzed data and the model observed during the training phase. Thus, we can automatically set this parameter and IT personnel can later adjust it as necessary.

Protocol		POSEIDON	POSEIDON+ ATLANTIDES
HTTP	DR	100%	100%
	FP	1683 (2,83%)	774 (1,30%)

Table 1: Comparison between POSEIDON stand-alone and POSEIDON in combination with ATLANTIDES using data set A; DR stands for detection rate (attack instance percentage), while FP is the false positive rate (packets and corresponding percentage); ATLANTIDES reduces false positives by more than 50% without affecting the detection rate (i.e., without introducing false negatives).

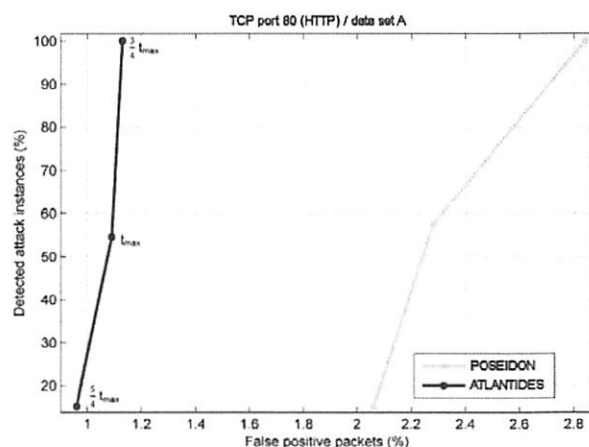


Figure 2: Detection rates for POSEIDON in combination with ATLANTIDES using data set A (HTTP protocol): the x-axis and y-axis present false positive rate (packets) and detection rate (attacks instances) respectively. It is possible to observe that ATLANTIDES presents a lower false positive rate than POSEIDON, considering the same detection rate. It is possible to notice how different ATLANTIDES' threshold settings affect detection and false positive rates.

Experiments and Results

To validate our architecture, we benchmark ATLANTIDES in combination with the signature-based NIDS Snort [34, 37] as well as ATLANTIDES in combination with the anomaly-based NIDS POSEIDON [5]. To carry out the experiments, we employ two different data sets. First, we benchmark the system using a private data set. Secondly, we use the DARPA 1999 data set [22]: despite criticism [23, 25] this is a standard data for benchmarking NIDSs (see, e.g., [33, 42]) and it has the advantage that it allows one to compare experiments. No other data set,

containing sufficient data to perform verifiable benchmarks, is publicly available.

We consider an attack to be successfully detected when at least one packet carrying the attack payload is correctly flagged as malicious; all the other non-detected packets carrying the attack payload are not considered to be false negatives. On the other hand, each packet incorrectly flagged as malicious is considered to be a false positive. Thus, the detection rate is attack-based, while the false positive rate is packet-based.

Tests With a Private Data Set

To carry out our validation, and to see how the system behaves when trained with a data set that was not made attack-free,³ we consider a private data set we collected at the University of Twente: this is *data set A*. Data were collected on a public network for five consecutive working days (24 hours per day), logging only TCP traffic directed to (and originating from) a heavy-loaded web server (about 10 Gigabytes of total traffic per day). This web server hosts the department official web sites as well as student and research staff personal web pages: thus, the traffic contains different types of data such as static and dynamically generated HTML pages and, especially in the outgoing traffic, common format documents (e.g., PDF) as well as raw binary data (e.g., software executables). We did not inject any artificial attack.

We focus on HTTP traffic because nowadays Internet attacks are mainly directed to web servers and web-based applications [17]: Kruegel, et al. [19] state that web-based attacks account for 20%-30% from 1999 to 2004 in CVE entries [39]; Symantec Corporation [38] reports that, in the first-half of year 2006, 69% of total discovered vulnerabilities were related to web services and, during the same period, more than 60% of *easily exploitable vulnerabilities* (whenever the exploitation code is not needed or well-known) affected web applications. Symantec states that typical examples of easily exploitable vulnerabilities are SQL Injection and Cross-Site Scripting (XSS) attacks.

To train the anomaly-detection engines of both POSEIDON and the OAD on data set A, we used a snapshot of the data collected during *working hours* (approximately three hours, 1.8 Gigabytes of data, randomly chosen). The chosen training data set had not been pre-processed and made attack-free: thus it is possible that the model includes some malicious activity (that could negatively affect accuracy). For the same reason, we randomly chose another snapshot (approximately 1.8 Gigabytes of data) to benchmark POSEIDON stand-alone against POSEIDON in combination with ATLANTIDES.

³This is useful to see how the system performs in the sub-optimal situation in which the IT security specialist does not have the time to clean up the training data set, a situation that is likely to occur often in practice.

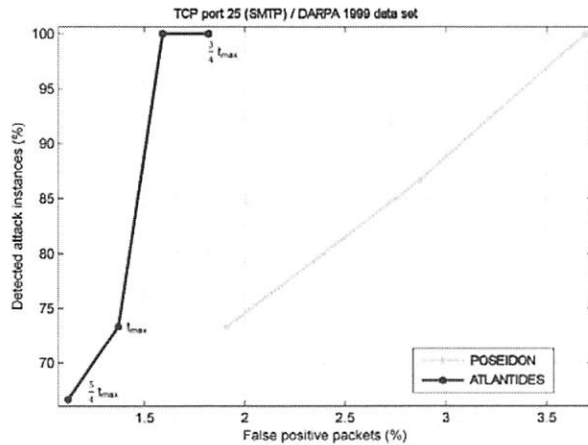


Figure 3: Detection rates for POSEIDON in combination with ATLANTIDES using DARPA 1999 data set (SMTP protocol): the x-axis and y-axis present false positive rate (packets) and detection rate (attacks instances) respectively. Is it possible to observe that ATLANTIDES presents a lower false positive rate than POSEIDON, considering the same detection rate. It is possible to notice how different ATLANTIDES' threshold settings affect detection and false positive rates.

ABSs can, obviously, achieve a 100% detection rate using a very low threshold value, but this negatively affects the false positive rate too (as we mentioned in Section Anomaly-Based Systems): we set the threshold of POSEIDON experimentally to achieve the best detection rate at the lowest false positive rate possible.

The alerts have been classified by the authors: we found evidences of XSS and SQL Injection attacks [43] (and this is not surprising, accordingly to Symantec's report), plus some probes checking for well-known paths (33 attack detections in total). Table 1 summarizes the results we obtained. We cannot compare ATLANTIDES in combination with Snort on data set A for the reason that Snort does not find any true attack to the system (Snort raised only false alerts):

this is not surprising, since Snort has only few signatures devoted to SQL Injections and XSS attacks. By setting a high threshold value in ATLANTIDES we could remove *all* the false positives, but this would give no indication of the completeness and accuracy of ATLANTIDES. Figure 2 shows detailed results of ATLANTIDES on data set A. Here, left is better than right and above is better than below. A point left-top indicates a configuration in which (almost) every attack has been correctly forwarded, with very few false positives left. On the other hand, a point on the low-right side indicates a configuration in which some real attacks have been incorrectly suppressed and a good deal of licit traffic was marked anomalous.

Tests With the DARPA 1999 Data Set

The testing environment of the DARPA 1999 data set contains several internal hosts that are attacked by both external and internal attackers: in our tests, we consider only inbound and outbound TCP packets that belong to attack connections against hosts inside the network 172.16.0.0/16. We focus on FTP, Telnet, SMTP and HTTP protocols. This is due to the fact that only these protocols, among the ones contained in this data set, provide us with a sufficient number of samples to train the OAD and, at the same time, allow us to compare our architecture with POSEIDON stand-alone, that has been benchmarked following the same procedures.

We train the OAD of ATLANTIDES with the data of weeks 1 and 3 (attack-free): for each different protocol we use a different OAD instance. Afterwards, we test ATLANTIDES together with both POSEIDON and Snort using week 4 and week 5 traffic. In order to distinguish between true and false positives, we refer to the attack instance table provided by the DARPA data set authors. Table 2 reports a comparison of the detection and false positive rates of Snort stand-alone (first column), Snort in combination with ATLANTIDES (second column), POSEIDON stand-alone (third column) and POSEIDON in combination with ATLANTIDES (fourth column).

Protocol		Snort	Snort+ ATLANTIDES	POSEIDON	POSEIDON+ ATLANTIDES
HTTP	DR	59.9%	59.9%	100%	100%
	FP	599 (0.069%)	5 (0.00057%)	15 (0.0018%)	0 (0.0%)
FTP	DR	31.75%	31.75%	100%	100%
	FP	875 (3.17%)	317 (1.14%)	3303 (11.31%)	373 (1.35%)
Telnet	DR	26.83%	26.83%	95.12%	95.12%
	FP	391 (0.041%)	6 (0.00063%)	63776 (6.72%)	56885 (5.99%)
SMTP	DR	13.3%	—	100%	100%
	FP	0 (0.0%)	—	6476 (3.69%)	2797 (1.59%)

Table 2: Comparison between Snort stand-alone, Snort in combination with ATLANTIDES, POSEIDON stand-alone and POSEIDON in combination with ATLANTIDES using the DARPA 1999 data set: DR stands for detection rate (attack instance percentage), while FP is the false positive rate (packets and corresponding percentage); ATLANTIDES reduces false positives by more than 50% most of the times, being close to zero in 3 tests, without affecting the detection rate (i.e., without introducing false negatives).

In both cases, ATLANTIDES achieves a substantial improvement on the stand-alone system, neither affecting the detection rate nor introducing false negatives; ATLANTIDES reduces the false positive amount by at least 50% on every protocol benchmarked, except for the Telnet protocol together with POSEIDON. In our opinion, this discrepancy is due to the fact that Telnet has a great output variability, since an user could issue hundreds of different commands with different output; on the other hand, protocols like HTTP, FTP and SMTP present well-defined protocol schemas to exchange information between client and server. ATLANTIDES is not applied to SMTP traffic in combination with Snort because in this case Snort raises no false positives.

Related Work

The problem of alert verification has been addressed using two different kinds of approaches: we have techniques for *identifying true positives*, and techniques for *identifying false positives*. The main difference between our work and the papers described below is that we take into account the outgoing traffic of the system.

Identifying True Positives

Kruegel and Robertson [18] introduces a plug-in for Snort to verify alerts: the plug-in integrates the Nessus vulnerability scanner into the Snort's core. When an alert is fired, this is not immediately forwarded but is firstly passed to the verification engine. Since every Snort's signature comes with a unique identifier (assigned by CVE [39]), this index is used to check the presence of a corresponding Nessus attack script. If found, the script is executed against the target machine/network: the output is extracted and used to flag the alert as either true or false; an output cache is used to avoid further verification for the same alert/target. Although this approach is effective, there are several drawbacks: one has to maintain the Nessus's attack script database updated, and this approach works only for signature-based NIDSs, while ATLANTIDES can work with both types and in a complete automatic way (i.e., no manual updates needed).

Ning, et al. developed a model [30] and an intrusion-alert correlator [27] to help human analysts during the alert verification phase. This work is based on the observation that most attacks consist of several related stages, with the early stages preparing for the later ones. Hyper-alert correlation graphs are used to represent correlated alerts in an intuitive way. However, this correlation technique is ineffective when attackers use a different (yet not spoofed) IP source address at each attack step. Ning and Cui [27] demonstrate the effectiveness of this approach when applied on a small data set (due to the exponential complexity of hyper-alert graphs): in [28, 29] the same authors present other utilities they developed to facilitate the analysis of large sets of correlated alerts, and report some benchmarks

employing network traffic used during the DEFCON 8 Capture the Flag (CTF) event [12]. ATLANTIDES does not present the same limitations on data set size.

Lee and Stolfo [20] develop a hybrid network and host-based framework based on data mining techniques, such as sequential patterns mining and episodes rules, to address the problem of improving attack detection while maintaining a low false positive rate. The system detects attacks combining different models and comparing them with actual traffic features. Benchmarks have been conducted using the DARPA 1998 data set [21]: detection score for different attack typologies has a minimum value of 65% with a false positive rate always below 0.05%. Since the authors use a different data set, we cannot compare directly the two approaches: however, we can notice that our approach does not use information collected from the operating system hosting the monitored network service(s), thus ATLANTIDES can work on-line without affecting the host performance.

Identifying False Positives

Pietraszek [33] tackles the problem of reducing false positives by introducing an alert classifier system (ALAC, Adaptive Learner for Alert Classification) based on machine learning techniques. During the training phase, the system classifies alerts into true and false positives, by attaching a label from a fixed set of user-defined labels to the current alert. Then, the system computes an extra parameter (called *classification confidence*) and presents this classification to a human analyst. The analyst's feedback is used to generate training examples, used by the learning algorithm to build and update its classifiers. After the training phase, the classifiers are used to classify new alerts. To ensure the stability of the system over time, a sub-sampling technique is applied: regularly, the system randomly selects n alerts to be forwarded to the analyst instead of processing them autonomously. This approach relies on the analyst's ability to classify alerts properly and on his availability to operate in real-time (otherwise the system will not be updated in time); we believe that these (demanding) requirements can be considered acceptable for a signature-based NIDS (where the analyst can easily inspect both the signature and network packet(s) that triggered the alert), but it could be difficult to perform the same analysis with an anomaly-based NIDS. Benchmarks conducted over the 1999 DARPA data set, using Snort to generate alerts, show an overall false positive reduction of over 30% (details on single attack protocols are not given).

The main differences between ALAC and ATLANTIDES include: (a) ALAC does not consider the outgoing traffic, and (b) ALAC relies heavily on the expertise and the presence of an analyst (in ATLANTIDES, all the IT specialist has to do is to set the thresholds).

Julisch [15] presents a semi-automatic approach, based on techniques which discover frequently occurring episodes in a given sequence, for identifying false positives based on the idea of *root cause*: an alert root cause is defined as “the reason for which it occurs.” The author observes that in most environments, it is possible to identify a small number of highly predominant (and persistent) root causes: thereby removing such root causes drastically reduces the future alert rate. Benchmarks conducted on a log trace from a commercial signature-based NIDS deployed in a real network show a reduction of 87% of false positives. No further details are given about the testing condition, network topology or traffic typology. We cannot compare directly this approach with ATLANTIDES because the data used by the author is private, nevertheless we can notice that this approach is applicable only to signature-based NIDSes, while ATLANTIDES is effective with anomaly-based systems too.

Analyzing output traffic The idea of analyzing (and correlating) the output of a (possible) compromised system as been used before in the context of worm detection.

Gu, et al. [14] scan the output traffic for specific port numbers. When an anomaly has been detected in the incoming traffic directed to a certain destination service port, their system start monitoring the output traffic to check whether the host tries to contact other systems using the same destination service port: if this is the case then the system is probably infected by a worm. Wang, et al. [41] proceed in a similar way, comparing outgoing to incoming traffic, looking for similarities: when an anomaly has been detected in the incoming traffic, the anomalous traffic is cached and compared to subsequent outgoing traffic (to detect polymorphic worms). A successful match indicates that the host has been infected and that the worm is trying to replicate itself, infecting other hosts. Any other kind of attack will not be handled by the system. In contrast, our solution presents a general architecture to carry out a complete anomaly detection on the output to reduce false positives of any NIDS placed on the input channel. Indeed we have shown that our architecture works well in combination with both a signature and an anomaly-based input NIDS.

Conclusion

In this paper we present ATLANTIDES, an architecture for automatic alert verification exploiting in a structural way the detection of anomalies in the output traffic of a system. ATLANTIDES can be used for reducing false positives both in signature and anomaly-based NIDSes. The core of ATLANTIDES consists of an output anomaly detector (OAD), which compares output traffic with a model it has created during the training phase. To reduce *false positives* on the input NIDS (be it signature or anomaly-based) monitoring the incoming traffic, ATLANTIDES checks if the communication raising an alert in the input NIDS

actually produces an anomaly in the outgoing traffic too. In this case (and in another exceptional situation), the alert is forwarded to the IT specialist, otherwise it is discarded. The fact that the OAD is anomaly-based (rather than signature-based) allows it to adapt to the specific network environment/service, and to work in an unsupervised way (at least, after the setup). Anomaly-based systems typically use a distance function and a threshold to discern anomalous from licit traffic. We introduce a simple heuristic to set ATLANTIDES threshold in an automatic, though effective, way, to further ease the management for IT security specialists (which can in case adjust the threshold value).

Benchmarks on a private data set and on the DARPA 1999 data set show that ATLANTIDES determines a reduction of false positives between 50% and 100% in most of the cases, without introducing any extra false negative, easing significantly the management of NIDSes.

One possible extension to our architecture is adding additional information to make the detection of anomalies in the output more precise: this information (e.g., the usual amount of bytes sent back from the server and the communication duration) could be included in the model and evaluated as well. Our architecture has been designed to work with TCP-based network services: although it could be easily adapted to work with UDP-based services, there exist some issues related to this protocol. In fact UDP is a connection-less protocol and this add some difficulties to distinguish real connections from the ones using spoofed IP addresses. We will investigate this in future.

Author Information

Damiano Bolzoni is currently a Ph.D. student at the University of Twente, Netherlands. His research interests are focused on intrusion detection systems and information risk management. He received a MSc in Computer Science from the University of Venice, Italy, with a thesis about anomaly-based network intrusion detection systems. He can be reached at dami.no.bolzoni@utwente.nl.

Bruno Crispo is a faculty member at the University of Trento and at the Vrije Universiteit Amsterdam. His research interests are security protocols, authentication, authorization and accountability in large distributed systems, RFID and sensors security. He has a Ph.D. in Computer Science from the University of Cambridge, UK. Contact him at crispo@dit.unitn.it.

Sandro Etalle received a Ph.D. from the University of Amsterdam, and has worked for the universities of Genova, Amsterdam, Maastricht, Trento. At the moment he is associate professor in the Distributed and Embedded Systems Group at the University of Twente, the Netherlands. His research covers trust management, intrusion detection systems and information risk management. He can be reached at sandro.etalles@utwente.nl.

Bibliography

- [1] Allen, J., A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner, "State of the Practice of Intrusion Detection Technologies," Technical Report CMU/SEI-99TR-028, Carnegie-Mellon University – Software Engineering Institute, Jan, 2000.
- [2] Axelsson, S., "Intrusion Detection Systems: A Survey and Taxonomy," *Technical Report 99-15*, Chalmers University, Mar, 2000.
- [3] Axelsson, S., "The Base-Rate Fallacy and the Difficulty of Intrusion Detection," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 3, Num. 3, pp. 186-205, 2000.
- [4] Bace, R., *Intrusion detection*, Macmillan Publishing Co., Inc., 2000.
- [5] Bolzoni, D., E. Zambon, S. Etalle, and P. Hartel, "POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System," *Proceedings of the 4th IEEE International Workshop on Information Assurance (IWIA)*, pp. 144-156, IEEE Computer Society Press, 2006.
- [6] Chaboya, D. J., R. A. Raines, R. O. Baldwin, and B. E. Mullins, "Network Intrusion Detection: Automated and Manual Methods Prone to Attack and Evasion," *IEEE Security and Privacy*, Vol. 4, Num. 6, pp. 36-43, 2006.
- [7] Check Point Software Technologies, *Stateful Inspection Technology*, 2005, http://www.checkpoint.com/products/downloads/Stateful_Inspection.pdf.
- [8] Clifton, C. and G. Gengo, "Developing Custom Intrusion Detection Filters Using Data Mining," *Proceedings of the 21st Century Military Communications Conference (MILCOM)*, Vol 1, pp. 440-443, IEEE Computer Society Press, 2000.
- [9] Dain, O., and R. Cunningham, "Fusing Heterogeneous Alert Streams into Scenarios," *Proceedings of the Workshop on Data Mining for Security Applications, 8th ACM Conference on Computer Security (CCS)*, pp. 1-13, ACM Press, 2002.
- [10] Debar, H., M. Dacier, and A. Wespi, "Towards a Taxonomy of Intrusion-Detection Systems," *Computer Networks*, Vol. 31, Num. 8, pp. 805-822, 1999.
- [11] Debar, H., M. Dacier, and A. Wespi, "A Revised Taxonomy of Intrusion-Detection Systems," *Annales des Télécommunications*, Vol. 55, Num. 7-8, pp. 361-378, 2000.
- [12] DEFCON8, *Defcon Capture the Flag (CTF) Contest*, 2000, <http://www.defcon.org/html/defcon8/defcon-8-post.html>.
- [13] Forrest, S. and S. A. Hofmeyr, "A Sense of Self for Unix Processes," *Proceedings of the 17th IEEE Symposium on Security and Privacy (S&P)*, pp. 120-128, IEEE Computer Society Press, 2002.
- [14] Gu, G., M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley, "Worm Detection, Early Warning and Response Based on Local Victim Information," *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pp. 136-145, IEEE Computer Society, 2004.
- [15] Julisch, K., "Mining Alarm Clusters to Improve Alarm Handling Efficiency," *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, pp. 12-21, ACM Press, 2001.
- [16] Julisch, K., "Clustering Intrusion Detection Alarms to Support Root Cause Analysis," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 6, Num. 4, pp. 443-471, 2003.
- [17] Klein, D. V., "Defending Against the Wily Surfer-Web-based Attacks and Defenses," *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, pp. 81-92, USENIX Association, 1999.
- [18] Kruegel, C. and W. Robertson, "Alert Verification: Determining the Success of Intrusion Attempts," *Proceedings of the 1st Workshop on the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2004.
- [19] Kruegel, C., G. Vigna, and W. Robertson, "A Multi-model Approach to the Detection of Web-based Attacks," *Computer Networks*, Vol. 48, Num. 5, pp. 717-738, 2005.
- [20] Lee, W. and S. J. Stolfo, "A Framework for Constructing Features and Models for Intrusion Detection Systems," *ACM Transactions on Information and System Security*, Vol. 3, Num. 4, pp. 227-261, 2000.
- [21] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman, "Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation," *Proceedings of the 1st DARPA Information Survivability Conference and Exposition (DISCEX)*, Vol. 2, pp. 12-26. IEEE Computer Society Press, 2000.
- [22] Lippmann, R., J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 DARPA Off-line Intrusion Detection Evaluation," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 34, Num. 4, pp. 579-595, 2000.
- [23] Mahoney, M. V. and P. K. Chan, "An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection," In *Proceedings of the 6th Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 2820 of LNCS, pp. 220-237, Springer-Verlag, 2003.
- [24] Manganaris, S., M. Christensen, D. Zerkle, and K. Hermiz, "A Data Mining Analysis of RTID Alarms," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 34, Num. 4, pp. 571-577, 2000.
- [25] McHugh, J., "Testing Intrusion Detection Systems: a Critique of the 1998 and 1999 DARPA

- Intrusion Detection System Evaluations as Performed by Lincoln Laboratory," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 3, Num. 4, pp. 262-294, 2000.
- [26] Morin, B., L. Mé, H. Debar, and M. Ducassé, "M2D2: A Formal Data Model for IDS Alert Correlation," *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 2516 of LNCS, pp. 115-127, Springer-Verlag, 2002.
- [27] Ning, P. and Y. Cui, "An Intrusion Alert Correlator Based on Prerequisites of Intrusions," *Technical Report TR-2002-01*, North Carolina State University, 2002.
- [28] Ning, P., Y. Cui, and D. Reeves, "Analyzing Intensive Intrusion Alerts via Correlation," *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 2516 of LNCS, pp. 74-94, Springer-Verlag, 2002.
- [29] Ning, P. Y., Cui, D. Reeves, and D. Xu, "Techniques and Tools for Analyzing Intrusion Alerts," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 7, Num. 2, pp. 274-318, 2004.
- [30] Ning, P., D. Reeves, and Y. Cui, "Correlating Alerts Using Prerequisites of Intrusions," *Technical Report TR-2001-13*, North Carolina State University, 2001.
- [31] Ning, P., and D. Xu, "Learning Attack Strategies From Intrusion Alerts," *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, pp. 200-209, ACM Press, 2003.
- [32] Paxson, V., "Bro: a System for Detecting Network Intruders in Real-time," *Computer Networks*, Vol. 31, Num. 23-24, pp. 2435-2463, 1999.
- [33] Pietraszek, T., "Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection," *Proceedings of the 7th Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 3224 of LNCS, pp. 102-124, Springer-Verlag, 2004.
- [34] Roesch, M., "Snort - Lightweight Intrusion Detection for Networks," *Proceedings of the 13th USENIX Conference on System Administration (LISA)*, pp. 229-238, USENIX Association, 1999.
- [35] Tenable Network Security, *Nessus Vulnerability Scanner*, 2002, <http://www.nessus.org/>.
- [36] Sommer, R., and V. Paxson, "Enhancing Byte-level Network Intrusion Detection Signatures With Context," *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pp. 262-271, ACM Press, 2003.
- [37] Sourcefire, Snort Network Intrusion Detection System Web Site, 1999, <http://www.snort.org>.
- [38] Symantec Corporation, *Internet Security Threat Report*, 2006, <http://www.symantec.com/enterprise/threat-report/index.jsp>.
- [39] The MITRE Corporation, *Common Vulnerabilities and Exposures Database*, 2004, <http://cve.mitre.org>.
- [40] Van Trees, H. L., *Detection, Estimation and Modulation Theory, Part I: Detection, Estimation, and Linear Modulation Theory*, John Wiley and Sons, Inc., 1968.
- [41] Wang, K., G. Cretu, and S. J. Stolfo, "Anomalous Payload-based Worm Detection and Signature Generation," *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 3858 of LNCS, pp. 227-246, Springer-Verlag, 2005.
- [42] Wang, K. and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," *Proceedings of the 7th Symposium on Recent Advances in Intrusion Detection (RAID)*, Vol. 3224 of LNCS, pp. 203-222, Springer-Verlag, 2004.
- [43] Web Application Security Consortium, *Web Security Threat Classification*, 2005, <http://www.webappsec.org/projects/threat/>.
- [44] Zhou, J., A. J. Carlson, and N. Bishop, "Verify Results of Network Intrusion Alerts Using Lightweight Protocol Analysis," *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pp. 117-126, IEEE Computer Society, 2005.

ATLANTIDES Pseudo-code

In this section we give a semi-formal description of how ATLANTIDES works.

DATA TYPE

$l = \text{length of the longest packet payload}$

$\text{PAYLOAD} = \text{array } [1..l] \text{ of } [0..255]$

/ packet payload */*

$\text{HOMENET} = \text{set of IP addresses}$

/ hosts inside the monitored network */*

$\text{HOST} = \text{RECORD } [$

$\text{address: IP address} \in \mathbb{N}$

$\text{port: TCP port} \in \mathbb{N}$

$]$

$\text{PACKET} = \text{RECORD } [$

source: HOST

destination: HOST

payload: PAYLOAD

$]$

$\text{alert} = \text{RECORD } [$

alert:

$-\infty$ if input NIDS is SBS

$\text{value} \in \text{Real}$ if input NIDS is ABS

$\text{processed: BOOLEAN}$

/ tracks a processed alert by the OAD */*

$\text{true_alert: BOOLEAN}$

/ alert is marked as an incident */*

$]$

DATA STRUCTURE

$\tau \in \mathbb{N}$

/ number of packets used for OAD training */*

$\text{oad} \in \text{NIDS}$

/ ABS analyzing outgoing network traffic */*

$\text{out_threshold} \in \text{Real}$

/ OAD threshold */*

$t \in \mathbb{N}$

/ time value to wait for output */*

$\text{pre-alerts} = \text{set of alerts}$

/ alerts received from the NIDS monitoring incoming traffic */*

INIT PHASE

/ IT specialists set out_threshold and t values */*

TRAINING PHASE

INPUT:

$p: \text{PACKET}$

/ outgoing network packet */*

for $t := 1$ to τ

/ first, train the OAD with τ samples */*

$\text{oad.train}(p.\text{source.address}, p.\text{source.port}, p.\text{payload})$ */* POSEIDON builds a profile for each monitored service */*

end for

TESTING PHASE

INPUT:

$p: \text{PACKET}$

/ outgoing network packet */*

OUTPUT:

$\text{true_alerts: set of alerts}$

for each $a \in \text{pre-alerts}$ do

/ checks if the packet belongs to a communication marked as anomalous by the input NIDS */*

 if $(\text{match_alert}(a, p) = \text{TRUE})$ then

$\text{anomaly_score} := \text{oad.test}(p.\text{source.address}, p.\text{source.port}, p.\text{payload})$

/ tests if the output is anomalous */*

 if $(\text{anomaly_score} > \text{out_threshold})$ then

$a.\text{true_alert} := \text{TRUE}$

$\text{true_alerts.add}(a)$

 end if

$a.\text{processed} := \text{TRUE}$

 end if

end for


```
for each  $a \in pre - alerts$  do                                /* missing-output-response handling */  
  if (a.processed = FALSE) and (current_time > t) then  
    a.true_alert := TRUE  
    a.processed := TRUE  
    true_alerts.add(a)  
  end if  
end for
```

PDA: A Tool for Automated Problem Determination

Hai Huang, Raymond Jennings III, Yaoping Ruan, Ramendra Sahoo, Sambit Sahu, and Anees Shaikh – IBM T. J. Watson Research Center

ABSTRACT

Problem determination remains one of the most expensive and time-consuming functions in system management due to the difficulty in automating what is essentially a highly experience-dependent task. In this paper we study the characteristics of problem tickets in an enterprise IT infrastructure and observe that most of the tickets come from very few products and modules, and OS problems present higher resolving duration. We propose PDA, a problem management tool that provides automated problem diagnosis capabilities to assist system administrators in solving real-world problems more efficiently. PDA uses a two-level approach of proactive, high-level system health checks, coupled with rule-based “drill-down” probing to automatically collect detailed information related to the problem. Our tool allows system administrators to author and customize probes and rules accordingly and share across the organization. We illustrate the usage and benefits of PDA with a number of UNIX problem scenarios that show PDA is able to quickly collect key information through its rules to aid in problem determination.

Introduction

Computer system administrators (SAs) play a number of important roles in managing enterprise IT infrastructure, either as members of internal IT departments, or with IT service providers who remotely manage systems for customers. In addition to handling installation, monitoring, maintenance, upgrades, and other tasks, one of the most important jobs of SAs is to diagnose and solve problems.

In IT services environments, the *problem management process* (defined, for example, in ITIL [4]) describes the steps through which computer problems are reported, diagnosed, and solved. A typical sequence is for a problem ticket to be opened by a call to the customer helpdesk, or by an alert generated by a monitoring system. This is followed by some basic diagnosis by first level support personnel based on, for example, well-documented procedures. Simple issues such as password resets or file restoration can often be handled here without progressing further.

If the problem needs further investigation, it is passed to second or third level personnel, who are typically SAs with more advanced skills and knowledge. They often start with vague or incomplete descriptions of problems (e.g., “application is running very slowly,” “mail isn’t working,” or “CPU threshold exceeded”) which require significant investigation before the cause and solution are found. In the context of server support, administrators often consult monitoring tools that provide some specific system indicators, and then log in to the server to collect additional detailed information using system utilities. In the course of day-to-day problem management, this process is often the most time consuming and expensive task for SAs – it

is difficult to automate, and requires field experience and expert knowledge.

Unlike the first level support personnel, there is hardly any well-documented procedure one can refer to during this advanced problem management process. SAs usually rely on their own knowledge and experience to diagnose the root cause of the problem. Because of the complexity of the problems, it is a significant challenge to create a useful documentation about this process which can be referred by others. Especially in the environment where supporting teams are globally distributed, how to create and share this knowledge is a big challenge.

In this paper we describe Problem Determination Advisor (PDA), a system management tool for servers that provides health indicators and automated problem diagnosis capabilities to assist SAs in solving problems. PDA is intended to be used by second or third level SAs who diagnose and address problems that cannot be handled by first level support (i.e., helpdesk). PDA uses a two-level approach that provides high-level health monitoring of key subsystems, and *scoped probing* that collects additional details in an on-demand, rule-based fashion. This approach has the advantage of performing detailed “drill-down” probing only when it is relevant to the problem at hand, hence avoiding the overhead of collecting such data all the time. Moreover, PDA’s probes and problem determination rules are not determined arbitrarily; they are crafted based on an extensive empirical study of real problems that occur in practice and expert rules drawn from system administrator best practices. The rules are also customizable to better serve a particular environment or purpose. Our specific contributions include:

- *Characterization of server support problems:* using real problem tickets from a diverse group of servers in a large enterprise, we study the relative frequency of various types of problems, and determine categories of commonly occurring problems based on their nature and frequency, and study the relative difficulty of resolving different types problems.
- *Problem determination rules for scoped probing:* based on examination of actual problem descriptions and their solutions, tools and scripts used by SAs in practice, and knowledge capture from SAs, we develop a set of rules that determine how probes should be dispatched to automatically collect the information necessary to assist the diagnosis of various types of problems.
- *Extensible PDA tool architecture:* we codify problem determination rules and best practices in a tool that can be expanded as new rules are developed or as specific needs arise in different IT environments.

The measure by which most system management tools, and problem determination tools in particular, are judged is the reduction they enable in the time or effort needed to solve problems. As such, we illustrate the usage and benefits of PDA with a number of UNIX problem scenarios drawn from problem tickets, discussions with SAs, and system documentation. In these scenarios, SAs must know which information to collect and how to collect it (manually) in order to solve the problem. We show that PDA is able to quickly collect key information through its problem determination rules to aid in finding the cause, or in some cases pin-pointing the problem precisely. This allows expert SAs to solve problems more efficiently, and less experienced SAs to benefit from the diagnostic best practices codified in the tool.

Our current library of problem determination rules and probes is geared toward system software problems, for example on commercial and open source UNIX platforms. However, the general approach of PDA is applicable to other problem areas, particularly applications and middleware (which are a significant fraction of all problems). As we discuss in Section Problem Characterization, the bulk of reported problems are related to a relatively small number of specific problem areas, and this holds across problems related to applications, platform, networking, etc. Hence, developing rules based on best practices for the most commonly encountered problems is quite feasible. However, we expect that this model is more beneficial for IT services environment where similar platform and configuration co-exist. For heterogeneous environment such as universities, the best practices may vary from each other.

In the next section, we present the results of a characterization study of problem tickets which we use to inform our choice of system probes and the design of

problem determination rules. Section PDA Design and Implementation follows with a description of the PDA tool design and implementation. In Section Problem determination experiences with PDA we evaluate PDA's efficacy in the context of several specific problem scenarios. Section Related work briefly discusses some of the related work. We conclude the paper in Section Summary with a discussion of our ongoing work in the implementation and evaluation of PDA.

Problem Characterization

We begin with a characterization of real-life problems from a large, distributed commercial IT environment. The problems are drawn from an analysis of about 3.5 million problem tickets managed through a problem tracking system over a nine-month period. These tickets contain rich amount of information including a problem description, indication of the affected system, and metadata such as timestamps, severity, and identity of the SAs handling the tickets. In our analysis, we derive a number of statistical characteristics that allow us to better understand the different categories and characteristics of common problems. Specifically, we focus on:

- which applications contribute to the majority of the problems, and in particular whether a few applications are responsible for most of the observed problems
- what are the most common causes of application problems
- what portion of problems arise due to application vs. operating system-level issues
- how much time is spent in resolving different types of problems

Our objective in this section is to develop insights from the analysis of problem tickets in a real enterprise IT environment, and later use these insights to develop tooling for improving the efficiency of problem diagnosis and resolution.

Fields in Problem Tickets

We examine a number of attributes of each ticket including both structured attributes with well-defined values, and unstructured attributes which are mostly free-text. Structured attributes contain information such as a ticket's open and close time, incident occurrence date, SA user ID, and some enumerated problem characteristics such as problem type, product name, etc. Problem description and solution description are free-text. In this study, we are particularly interested in the following data fields:

- *product name:* There are about 600 products appearing in the tickets. Most of them are application names, while operating system or platform-related problems are categorized by general terms such as AIX, HP-UX, Linux, Windows.
- *product component:* Each product is further broken down into various predefined components.

They give finer-grained information about the problems. For example, components within the Windows product include bootup, explorer, system errors, password.

- *product module*: This is the finest-grain information available in the problem database – the module identifies the system sub-component that is having the problem. For example, in Windows, the bootup component is divided into different modules such as safe mode, unable to boot, inaccessible boot, and explorer contains navigation, move/copy Files, search.
- *ticket type*: When a ticket is first opened, it is categorized into a type, for example: error, performance, information request, load request and others. When studying common problems, we focus on those tickets with type of error and performance because they usually require further diagnosis to identify the root cause.
- *cause code*: This field has 34 values to describe various problem causes. It is particularly useful when other fields such as component and module are not specified. For example, all of the tickets related to the Linux product name have *software* as the component field and *OS* for module.

Problem Characteristics Overview

To understand the problem tickets in the system, we start from statistical characteristics of the tickets based on the above-mentioned fields. We make the following observations:

- **Most of the problem tickets arise from a few products.**

We examined all tickets according to the product name field. Among the 600 products, we observe that 50 products, which are less than 10% of the total products, account for 90% of all tickets. Figure 1 shows a cumulative frequency graph of the number of products and the percentage of the tickets with that product. This observation is similar to the failure characteristics observed on Windows XP machines [5]. Among the top 50 products with the most problem tickets, the number one product is an enterprise email system, followed by a virtual private network (VPN) application, and then a popular operating system. Figure 2 shows more detail about the number of tickets and their distribution, and Table 1 shows the top 10 products and the number of tickets from each.

- **Within each product, most of the problems come from a few modules.**

Next, we analyze the top products which have the most tickets by categorizing them according to product components and modules. Table 3 lists the 18 modules which comprise 70% of the tickets reported for the mail system, while the total number of modules defined for this product is

162. For the VPN application, we see an equally skewed distribution: 70% of the tickets are from six modules, out of a total of 70 modules. Table 2 lists these top six modules. Details of product components are not listed here because they reveal less information than the modules.

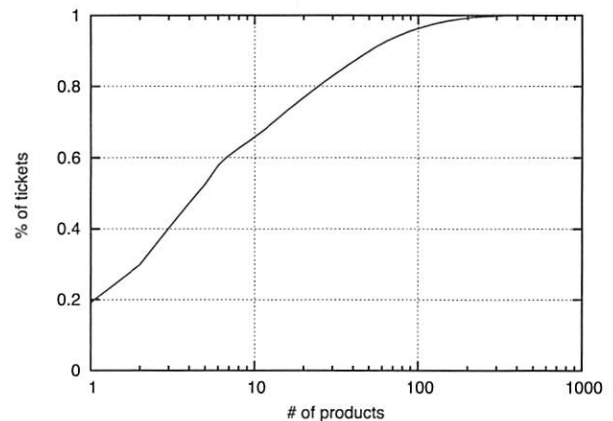


Figure 1: Cumulative frequency of the tickets contributed by the set of products.

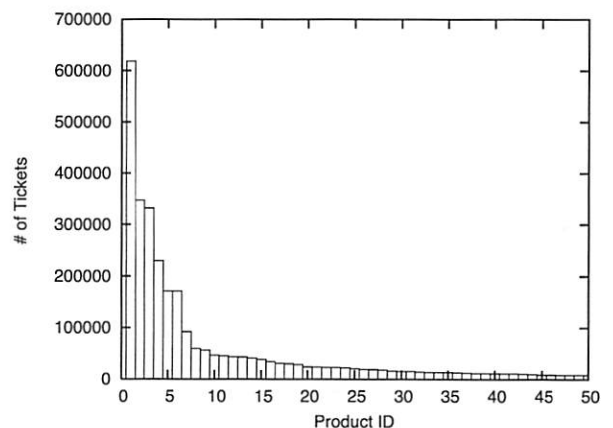


Figure 2: Number of tickets from the top 10% products with the most problem tickets.

These two observations together suggest that, by focusing on problem determination for a relatively small number of products and corresponding modules, we are able to cover a large portion of problem tickets. However, it is also important to note that each module may in fact exhibit many problem symptoms and possibly many root causes. This implies that a practical tool should be highly customizable in order to address symptoms that might be unique to a particular environment.

Operating System Problems

In terms of quantity of problem tickets, operating system (OS) problems are not as prominent (with the exception of Windows-related tickets, which consist of about 11% of the all tickets). However, system software or OS tickets are often the most diverse and require significant effort to diagnose. Figure 3 illustrates

the distribution of time spent in resolving problem tickets for top applications and systems/OSes. Our analysis indicates that nearly an order of magnitude more time is spent in the resolution of problems arising from UNIX system issues, than on other types of problems. VPN and mail applications, both comprising a large number of application-related problem tickets, need considerably less time to resolve compared to UNIX system related problems.

Product name	Cumulative # of tickets	% of tickets
Mail app	618575	18%
VPN app	346812	28%
Windows OS	331978	38%
Mail app (prev version)	229415	45%
PC hardware	171078	53%
Network connectivity	170937	58%
Software installer	92358	61%
Mainframe local app	59314	63%
Telephone	56396	64%
Desktop security audit tool	46470	66%

Table 1: Top 10 products in the ticket database.

Product name	Cumulative # of tickets	% of tickets
RESET	114477	38%
LOOPING	23620	45%
INTRANET APP	22045	53%
INFO	20537	60%
INTER/INTRA NET	14220	64%
CLIENT	13409	69%
UNABLETOCONNECT	13390	73%

Table 2: Product modules and the number of tickets from these modules for the VPN application.

We also examine the top problem types for OS platforms. Unfortunately, there are no OS components and modules defined except for Windows (perhaps due to the complexity of OS problems). Instead, we use the cause code field in analyzing OS tickets. We combine similar cause code values to arrive at a set of broader problem categories including: application, configuration, hardware, request/query (e.g., password reset or howto questions), duplicate (i.e., multiple reported problem), storage, network, human error, unsupported (i.e., out of scope for the support team). Note that “application” in the OS tickets differs from application as standalone product – these are mostly system processes or services shipped with the OS such as Sendmail, NFS, Samba etc.

Common OS Problems

From the problem categories in Figure 4, we further examine ticket details in a few categories to

identify commonly occurring problems that occur in each category. In particular, we focus on problem categories related to systems software and application-related issues on the UNIX platform, including: application, configuration, storage, and network. We use a combination of ticket clustering based on structured attributes, and manual inspection of problem and solution description text, to extract a set of typical problems. We describe a few of these sample problems below.¹ Recall that the problems are grouped by cause code values that are indications of the identified cause; this may be a different category than the original problem description would initially indicate.

Product name	Cumulative # of tickets	% of tickets
OPENING DB	54745	12%
RESET	34816	20%
FEATURES	32775	27%
INSTALL/SETUP	25365	32%
OSM/FILESIZE	23458	37%
OPTIONS	17788	41%
SETTINGS	17453	45%
SEND/RECEIVE	17013	49%
HELP	13785	52%
TEMPLATE	13049	55%
CHANGE	12229	57%
SETUP	10631	60%
CREATE ID	9348	62%
SENDING	8786	64%
OPTIONS/PREFS	8385	65%
NOTRESPONDING	7086	67%
GENINFO	7034	69%
AUTHORIZATION	6452	70%

Table 3: Product modules and the number of tickets from these modules for the mail application.

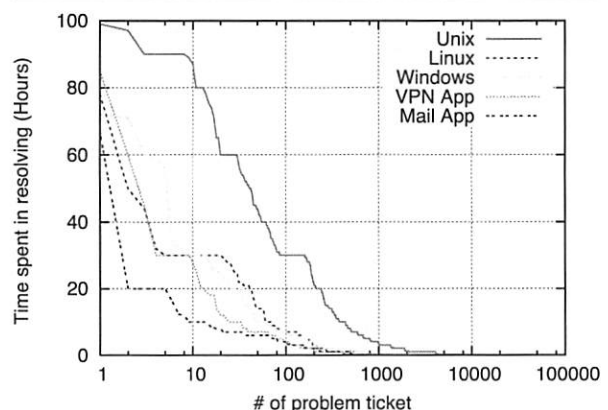


Figure 3: Time spent in solving tickets in top two products and in three OSes.

¹Specific hostnames, directory names etc. have been anonymized in these samples.

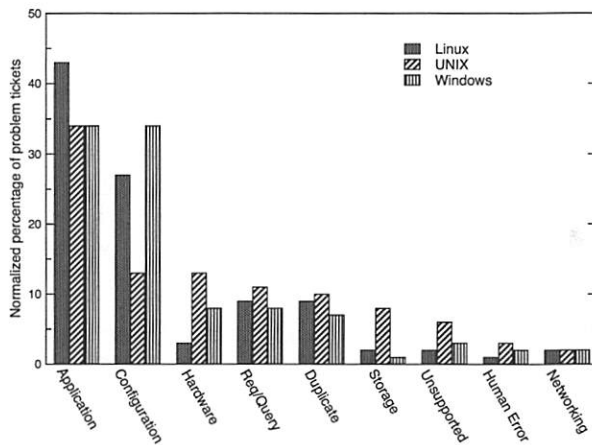


Figure 4: Categorization of problem tickets for Linux, UNIX and Windows servers. We examined 4,598, 42,998, and 331,978 tickets for each platform, respectively.

Configuration Problem Samples

- *Multiple email messages from dissimilar domains appear to be blocked.* The reason was due to a change in spam filtering mechanism/update.
- *SSH failing after OS version upgrade.* The solution of this problem was to change permissions on /dev/random and /dev/urandom to 644
- *The following NFS filesystems did not mount after miwsvr1 rebooted today:*
`efkxeastapps.abc.xyz.com:/u0`
`efkxeastapps.abc:/u1`
`efkxeastapps.abc.xyz.com:/u2`
`efkxeastapps.abc.xyz.com:/u3`
`efkxeastapps.abc.xyz.com:/u4`

The reason of this problem was because the filesystem has been updated using another type. Solution was to mount the filesystem using that type name.

Application Problem Samples

- *the databases will not start because of some TCP/IP problem.*
 The reason was because of port mapper down.
- *03:38 OPS received the following red alert on UNIX icon for abcdef03:*
`Host:named.my.ibm.com`
`Msg:The percentage of available swap space is low (20.12939453125 percent).`
 The solution was to stop unwanted processes.
- *SRM data is not getting retrieved from STI-BACKUP due to the refusal of the SFTP connection. Symptoms suggest SSH is not running.* The problem was solved by changing /usr/bin link to ssh and /etc/inetd.conf path then restarting ssh.

Storage

- *Base_OS_Monitors critical 98.006800 Percent space used (/var) This Critical Sentry2_0 disk*

usedpct event was received by a monitoring server at 3/6/2006 6:20 EST.

Typical solutions to this space usage problem include removing old files, large files, expanding file system space, and compressing the old files, etc.

- *System backup failed for sx0000e0 on 18082006. More details could be found in the logfile /var/adm/mksysb.out.*
 Reason for this problem was because some temporary files were not found.

A summary of examples of top problems that we found among problem tickets is shown in Table 4 in the first column.

Implications From Problem Ticket Analysis

Our observations in characterizing problems in a large IT environment has a number of implications that guide the design of PDA. Recall that the first key finding was that a few products or applications are responsible for the majority of problem tickets opened. The second observation was that the cause of these problems can be attributed to a relatively small set of functional components of these products. These results together imply that problem determination tooling that addresses a finite and fairly small set of important problem types can in fact cover a significant portion of problem tickets that are observed in practice.

We also observed that, in terms of time spent on resolution, UNIX system related problems are relatively difficult to resolve. Therefore, this is an important problem area to consider. Improving diagnosis efficiency for UNIX-related problem can potentially provide a significant value in terms of reduced time and effort.

These observations motivate our implementation of PDA, which addresses a number of key categories of system software or OS-related problems. While our intention is to broaden the applicability of PDA to other problem types (e.g., applications), our initial focus on OS problems on UNIX platforms is justified based on the analysis presented above.

PDA Design and Implementation

In this section, we first describe the overall design, architecture, and implementation of Problem Determination Advisor. We also describe how knowledge gathered from problem tickets is incorporated in PDA's automated problem diagnosis capabilities. Details of problem determination rules and system probes, as well as some realistic examples, are also illustrated below.

Design Overview

From the previous section, we have observed that a large percentage of problem tickets are related to a small number of products, and within each, most problems have only a few primary causes. We use a two-level approach that provides high-level health monitoring of key subsystems, and scoped probing

that collects additional system details. In Table 4, the second column lists some of the high-level monitors that are used in practice to identify the occurrence of common problems, while the third column shows examples of diagnostic probes that can help identify the cause of common problems in the corresponding category. The list of problems reported here represents a sample of the problem scenarios we have examined, and is by no means complete. We are continuing to expand the list of common problems and corresponding problem determination probes as we examine additional tickets and continue the knowledge capture of SA best practices.

If the full complement of monitors and probes are always active (e.g., executing periodically), they would likely impose a noticeable overhead on production systems. Hence, the two-level probing approach uses (i) periodic, low-overhead monitoring to provide a high-level health view of key subsystems, and (ii) detailed diagnostic probes when a problem is detected. This is similar to the way SAs tackle problems – the difference being that PDA tries to collect the relevant problem details automatically.

The knowledge of which diagnostic probes should be run when a problem is detected is encoded in problem determination rules. These rules are represented in a decision tree structure in which the traversed path through the tree dictates the series of diagnostic probes that are executed. At each node, the output of one or more diagnostic probes is evaluated against specified conditions to decide how to proceed in the traversal. In our implementation, diagnostic probes generally use available utilities on the platform directly to retrieve the needed information.

Problem Determination Rules

Figure 5 shows a sample rule tree which can diagnose problems related to the network connectivity of a managed server. The corresponding health monitor considers the system's network connection to be available if it is able to reach (e.g., ping, or retrieve a Web page)

several specified hosts outside of its subnet. These could be other servers it depends on, or well-known servers on the Internet, for example. If a disconnection is detected by the health monitor, scoped probing using diagnostic probes will be invoked to gather information to help determine the root cause of the problem. According to the rule tree, the first diagnostic probe should check the network stack by pinging the loop-back address. If no problem is found, the next node in the rule tree will dispatch another diagnostic probe to check that a default gateway is defined in the local routing table, and that it is reachable. If it is unreachable, the problem might be with the local subnet or network interface card. Otherwise, potential DNS-related problems are checked, for example verifying that `/etc/resolv.conf` exists, and that it contains DNS server entries (that are reachable). Clearly some diagnostic probes have dependencies (e.g., check `/etc/resolv.conf` exists before checking that a DNS server is reachable) and have to be executed in a certain order. In the absence of dependencies, probes could be ordered differently, perhaps tailored to the likelihood of certain types of failures in a given environment.

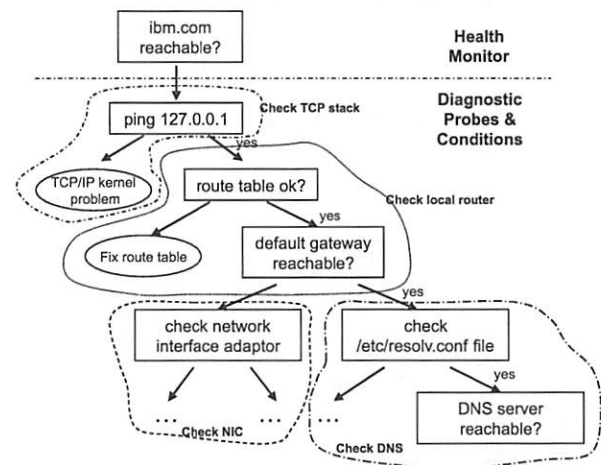


Figure 5: Sample rule for network-related problems.

Category	Typical Problems	Health Monitors	Diagnostic Probes
Configuration	OS/App upgrades, changes to various configuration files, firewall, spam filtering	Track changes to upgrades, configuration files	File privilege, active users, file diff with backup
Application	App errors due resource exhaustion (incl. CPU, memory, filesystem), app prerequisites, path/setup, etc.	Check resource usage, error log, process status etc.	Processes having most of the CPU, mem, IO, etc.
Network	Connectivity, performance	Check DNS, firewall, routing table, NIC	Traceroute, TCP dump, network options, NIC
Storage	Capacity, data corruption, mount problem, performance, disk swap, etc.	Check available space, I/O rate, error logs	Mount options IO history, big files

Table 4: Examples of common problem symptoms and corresponding monitors and probes used to identify these problems.

Our problem determination rules are primarily derived from inspection of problem tickets and by capturing best practices from SAs (i.e., through discussions, reviewing their custom scripts and procedures, etc.). In the case of problem tickets we extract rules by examining the steps through which a problem was diagnosed and resolved. When the detailed steps are available, creating a corresponding rule tree is fairly straightforward. Some of the tickets, however, do not have much detail beyond the original problem description and perhaps a few high-level actions taken by the SA. In such cases, we must manually infer the probes needed to collect the appropriate diagnosis data. In our rule extraction process, we use a combination of data mining tools to categorize problem tickets and identify distinguishing keywords, followed by varying degrees of manual inspection to better comprehend the problem and solution description text. The data mining tools are not described in detail here. Our experience with problem tickets so far leads us to believe that some amount of manual inspection is necessary to derive corresponding rules, however we continue to investigate automated techniques to assist the rule derivation process.

System Architecture

We have implemented a fully functional prototype of PDA, including the probing and data collection mechanisms, rule execution, and a Web-based interface. Figure 6 shows the overall architecture of PDA, which contains three major components: probe daemon, PDA

server, and the user interface. Our probe daemon is implemented in C for performance consideration and for easy deployment. PDA server is implemented in Java and our current user interface backend uses IBM WebSphere Portal Server. We use MySQL as our database storage.

Web User Interface

The Web UI allows SAs to perform various tasks from a single interface accessible from any workstation. It gives SAs an at-a-glance health overview of all of the servers being managed by clearly highlighting systems and components that have problems or are predicted to have problems in the near future. Whether or not an indicator implies a problem is determined by the corresponding rule, as discussed further below. In addition to showing the current health view of managed servers, we also allow SAs to look at the status of the servers at earlier points in time. This feature is useful when a reported problem is not currently evident on the system, but may be apparent when viewing system vitals collected earlier. It also is crucial for observing trends in certain metrics. Based on our discussions with SAs supporting commercial accounts, this feature is particularly useful to them in gaining a better understanding of the behavior of the managed systems.

Besides monitoring, the Web UI allows SAs to perform some simple administrative tasks such as adding another server to be monitored, updating a user's access privilege (as a root SA), adding or removing

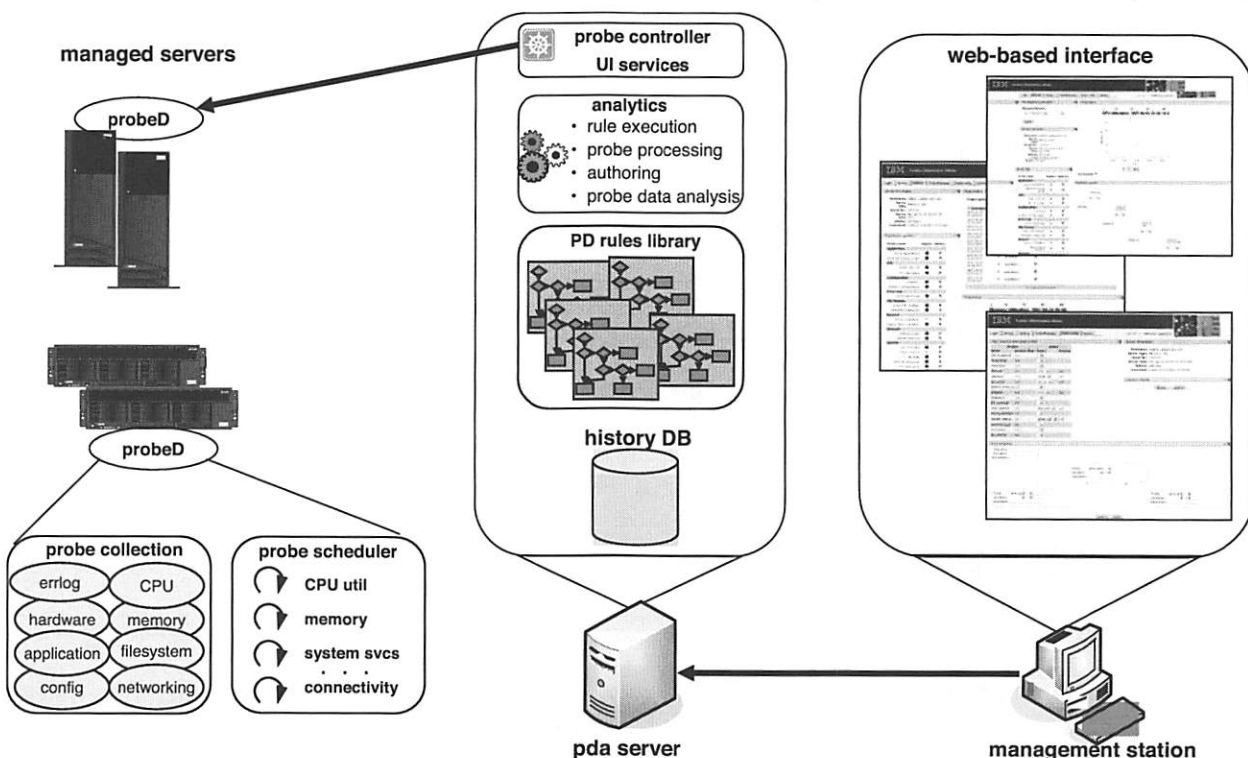


Figure 6: PDA architecture.

probes on a managed server, etc. The Web UI also provides an interface for SAs to author probes and construct rules from new and existing probes. Rules and probes constructed using this interface are stored in XML to facilitate sharing with other SAs or reusing them across multiple platforms. This feature is discussed in more detail in Rule Sharing Section.

We are also in the process of implementing a Web based secure login mechanism so that SAs can quickly switch from a shell from one machine to another using pre-defined credentials. This allows SAs to better visualize monitored data using the graphical interface, while still having access to a low-level command line interface from a single tool.

Probe Daemon

The probe daemon is a small program that runs on managed servers whose primary functions are to schedule the execution of probes according to the frequency set by the SA, and to interface with the PDA server. The PDA server sends command to start a new probe, stop an existing probe, change the periodicity of a probe, etc. When starting a new probe, the probe daemon can download the probe from a central probe repository if the probe does not exist or is not up-to-date locally. After probes have finished executing, the probe daemon is also responsible to send probe results back to the PDA server.

PDA Server and Rule Library

Most of the information exchange and processing is handled by the PDA server. Periodically it receives probe results from the probe daemon, stores the results in a history database, and triggers rule execution if there is a corresponding rule for a particular probe. The history database stores collected data and also serves as a repository for important configuration files that are tracked by PDA.

The rule execution engine is the most important part of the PDA server. It parses rules, each defined in a separate XML file located in the database, and converts them to an in-memory representation for evaluation. There are two ways to evaluate a rule. Typically, a rule is triggered by a periodic probe which is defined to be the root node of the rule tree. The trigger can be a threshold violation, change in a key configuration file, or other detected problem. As a second method, an SA can execute a rule to initiate it manually, to proactively collect information related to specific subsystem.

In both cases, as the rule tree is traversed, a command is sent to the probe daemon to execute the required probe and return the result. The result is used to make a decision to continue collecting more information or stop the rule execution. Some rules also use historical data to decide what should be collected next, or what should be displayed to the SA.

Since PDA supports management of multiple groups of servers (e.g., for different customers), the PDA server also keeps track of which servers are

managed by which SAs. This also implies that each server or group can have different active rules and probes; PDA supports this notion of *multi-tenancy* in the rules library and rule execution engine.

Rules and Probes

A probe is usually implemented as a script (e.g., Perl, shell, etc.) that either executes native commands available in the system or interfaces with other monitoring tools deployed in the environment. The probe parses and aggregates the output of the commands, and returns the results as an XML document. In order to make it easy to add new probes to PDA, the schema is a simple and generic key-value pair representation. When interfacing with other monitoring tools to collect data, we write adapters to convert their output to the XML format for the PDA server.

Figure 7 is an example output from a probe that monitors Ethernet interfaces.

Rules are triggered automatically by a monitoring probe which appears as the first node of the rule tree. For example, Figure 8 shows a sample rule *chk_interface*. It will be triggered by a probe called *chk_eth*. In this rule, the first step tests if the number of collisions is beyond a certain threshold. If the threshold is exceeded, the next probe, *chk_switch*, is executed to collect some information about the network switch, for example related to the firmware version. This type of scoped probing minimizes monitoring overhead and expedites the problem determination process. In the case where the probe in the first node does not present, or the problem is reported by other channels, such as problem ticket, a rule can be executed manually by the SAs.

```
<results probename="chk_eth">
  <result>
    <key> INTERFACE </key>
    <value> eth1 </value>
  </result>
  <result>
    <key> ERRORS </key>
    <value> 0 </value>
  </result>
  <result>
    <key> DROPPED </key>
    <value> 0 </value>
  </result>
  <result>
    <key> COLLISIONS </key>
    <value> 50234 </value>
  </result>
</results>
```

Figure 7: A sample probe output.

Probe and Rule Authoring

Given the heterogeneity of enterprise systems and applications, it is unrealistic to expect a single library of rules and probes to work in all IT environments. For this reason, PDA is designed to be

extensible to allow authoring of probes and rules, either from scratch or, more commonly, based on existing content. We provide templates for writing new diagnostic probes, and a way to logically group rules and their associated probes (e.g., based on a particular target application). Being able to quickly construct a rule for an observed problem from a set of existing probes can be very helpful to save SAs precious time.

We have implemented a Web-based probe and rule authoring interface. We validate the input of restricted fields and perform some simple checks (e.g., for duplicated names in the repository). Validation of the probe code is similarly simple, comprising checks that the probe runs successfully and implements the necessary rule output formatting. Newly created rules require slightly more involved validation. For example, we validate that each node in the rule tree has the corresponding probe(s) available, and that the conditions being checked are supported by the probe.

Once a new probe is authored, its data fields are created and stored directly in the corresponding tables in the database, and the PDA server is notified of the new probe name. If the probe is periodic and needs to be activated for monitoring, the probe daemon is contacted by the PDA server automatically to schedule the probe. If the probe is for diagnostics (i.e., a “one-time” probe vs. periodic), it will be downloaded to the managed server when it is invoked by a rule. Rules are stored similarly, along with the XML representation of the tree structure.

Since our probes are mostly scripts which will be running on managed servers, they pose a potential security threat to the system if probes are malicious. Currently we rely on user authentication and out-of-band change approval for new probe and rule authoring. It may also be feasible to use compilation techniques to perform some checks on the semantics of the scripts, for example to see if a probe is writing to a

restricted part of the filesystem. We are investigating this in our ongoing work on PDA.

Rule Sharing

In our discussions with SAs, we found that sharing knowledge and experience between them is a considerable challenge. One potentially significant benefit of rule and probe authoring in PDA is the opportunity to share them with other SAs managing the same environment, or even those working in very different environments.

Guaranteeing that rules and probes authored by one SA are applicable to problem resolution on other systems poses a number of difficulties. The primary one is the wide variety of platforms, operating systems, and software. Most rules are largely platform-independent as the information can be extracted on most OSes. However, the probes that actually collect the information can be quite different on various platforms. Even on machines with the same OS, different patch levels or software configurations can very easily break probes. To make sharing of rules and probes more seamless to SAs, we annotate them with dependency information that indicates the platform and version on which they have been deployed or tested.

Initially, we expect to deploy our tools in a fairly homogeneous environment, e.g., with mostly UNIX machines. We expect most dependency issues in such an environment to be solved relatively easily, for example by using a different binary/utility to obtain the same information. This technique can be carried over to managing other flavors of UNIX or Linux. As more users contribute rule and probe content over time, they will likely cover a more comprehensive set of platforms and provide a valuable way to accumulate and codify system management knowledge.

PDA Usage Model

The design of PDA is largely motivated by our experience in IT service provider environments, in

```
<rule rulename="chk_interface">
  <node probename="chk_eth" id="0">
    <condition> COLLISIONS > 500 </condition>
    <true-branch> id="1" </true-branch>
  </node>
  <node probename="chk_switch" id="1">
    <condition> MANUFACTURER == LINKSYS &&
      MODEL == ETHERFAST &&
      FIRMWARE_VERSION <= 2.3.1
    </condition>
    <true-branch>
      alert("upgrade firmware")
    </true-branch>
    <false-branch> id="2" </false-branch>
  </node>
  <node ...>
    ...
  </node>
</rule>
```

Figure 8: A sample rule file.

which globally distributed support teams manage the infrastructure belonging to a large enterprises. In these environments, creating, communicating, and adhering to best practices for systems architecture and management is a significant challenge. Global support teams often consist of administrators with greatly varying amounts of experience and knowledge – providing the ability to capture and operationalize problem determination procedures for the whole team is very valuable. In addition to improving efficiency through automated collection of relevant information, it also allows SAs to follow similar procedures which could be designed by the most experienced team members. At the same time, PDA allows customization of problem determination rules to account for different customer environments or priorities. A PDA installation could include a standard set of problem determinations rules and associated probes that handle common or general problems (e.g., networking problems, excessive resource consumption, etc.). These could be supplemented with new content that is available from a central repository, or through local modifications of existing content.

This usage model is particularly applicable for IT service providers who manage many customer infrastructures in a number of industries, since it is likely that there is a lot of similarity at the system software level. Furthermore, when the service provider has performed some degree of transformation in the customer environment, for example to move to preferred platforms and tools, the possibility for sharing and reuse increases. In IT environments belonging to universities and industry research labs, we observe more heterogeneity in OS platforms, applications, and usage, which makes it more difficult to develop problem determination best practices that are widely applicable. Nevertheless, the automation and extensibility features of PDA are still useful in these situations.

Problem Determination Experiences With PDA

This section describes some of our experiences in constructing rules and probes to diagnose practical problems with PDA. The rules we built based on problem tickets and best practices from interviewing SAs are not comprehensive, but they address commonly occurring problems in realistic settings and can be expanded and enhanced by communities of users or administrators.

Experience with NFS Problems

NFS allows files to be accessed across a network with high performance, and its relatively easy configuration process has made it very popular in large and small computing environments alike. However, when problems occur, finding the root cause can take a significant amount of time due to NFS's many dependencies. Most of the NFS-related problem tickets we observed are straight-forward to solve, but some have symptoms that are difficult to connect with their final solution. This often results in tickets being forwarded

multiple times to different support groups, sometimes incorrectly, before they are finally resolved. We found that most of these problems can be diagnosed with a few simple systematic steps.

```
$raw = `lssrc -a`;
$lines = split(" ", $raw);

# Omitted code for error-checking
#     executing lssrc

# Parses lssrc output: 3 possible
#     output formats
# 1. Subsystem Status
# 2. Subsystem Group Status
# 3. Subsystem Group PID Status

foreach $line (@lines)
{
    $line =~ s/^s+|s+$//g;
    @lineElem = split(/s+/, $line);
    $numLineElem = scalar(@lineElem);
    $match = $lineElem[0] eq $ARGV[0];
    if ($numLineElem == 4)
    {
        if ($match &&
            $lineElem[3] eq "active")
        {
            $foundActive = 1;
        }
        elsif ($numLineElem == 3)
        {
            if ($match &&
                $lineElem[2] eq "inoperative")
            {
                $foundButNotActive = 1;
            }
        }
        elsif ($numLineElem == 2)
        {
            if ($match &&
                $lineElem[1] eq "inoperative")
            {
                $foundButNotActive = 1;
            }
        }
    }

    # Format output and dump to stdout
    if (defined($foundActive))
    {
        &PDAFormatOutput(...);
    }
    elsif (defined($foundButNotActive))
    {
        &PDAFormatOutput(...);
    }
    else
    {
        &PDAFormatOutput(...);
    }
}
```

Figure 9: A simple probe that checks if a system service is currently running.

The rule to determine NFS-related problems is shown in Figure 10 as a tree. The rule tree is traversed by information gathered from dispatching a series of probes. Some rules are intuitive – check for liveness of all NFS service daemons, e.g., *nsfd*, *mountd*, *statd*, and *lockd*, and check if these services are properly registered with the portmapper. In Figure 9, we show an example probe that checks if a system service has been started and is currently running. As each probe does something very specific, it can be quickly and easily implemented and maintain. We used Perl to implement this probe, but probes can be written in any language as long as their output format matches the pre-defined format. There are also other more esoteric rules – *statd* should always start before *mountd*, or the *exname* option can only be used if the *nfsroot* option is specified in */etc/exports*. However, we have seen that a majority of problem tickets can be addressed by the simpler checks, e.g., looking for a missing */etc/exports* file, non-existent mount points, etc. Some probes are useful to exercise periodically to help SAs maintain a

healthy NFS service (e.g., check for hung daemons or abnormal RPC activities). Some can be run when changes are made to configuration files to check for potential problems caused by a recent change. Some can even be triggered manually in response to client-reported problems.

The benefits of PDA's automated problem determination capability are best illustrated by considering a specific problem scenario. Consider the problem of a misconfigured system that starts `nfsd` before `portmapper`. The misconfigured system will not allow users to access remote NFS partitions and the resultant problem ticket has a correspondingly vague problem description. In the absence of PDA, the SA must manually narrow down the problem cause, starting for example, by logging into the affected systems, checking network connectivity, verifying firewall rules, etc. to make sure the NFS problem is not a side effect of another problem in the system. Having done that, the SA may then check `/etc/exports` for possible permission problems, see if all the defined mount points exist and run the `ps` command to confirm that all NFS-related services are running. He or she might need to run more diagnostic tools before finally discovering that `nfsd` was not correctly registered with the `portmapper`. Using PDA however, this misconfiguration problem can be discovered quickly by examining the results of the automated rule execution.

Experience With Storage Problems

We were surprised that a large percentage (90%) of the storage-related problem tickets we observed were related to simple capacity issues, e.g., disk partitions used for storing temporary files or logs being close to or completely full, thus hampering normal operations in the system. Given the large volume of such tickets, early detection and resolution of storage capacity problems can potentially eliminate a large number of tickets. In many environments, management tools monitor filesystems and raise an alert if the

utilization crosses a specified threshold. This approach has the disadvantage of potential false alarms if for example, a filesystem is normally highly utilized (e.g., an application scratch area). In PDA, we use a low-overhead profiler that periodically samples disk usage for each partition being monitored and have the profiler raise an alert if a partition's usage (within a time interval N) has an upward trend that is projected to be completely filled within the next 24 hours. A 24-hour period is chosen to allow an SA enough time to confirm a problem and take action. An example illustrating how the profiler is able to find storage capacity problems is shown in Figure 11. In this example, simple linear regression is used for trending, which will detect that in interval 3 this partition will be full within a day and raise an alert. More complex regression methods can be used to further suppress false alarms.

As an example similar to what we described in Section 2.4, consider the actual problem ticket with this description: *Ops received the following alert on host: \$(<\$hostname>) \$ "Percent space used (/var/log) greater than 95% - currently 100%."* This problem ticket was opened by the operations team after an alert was raised by a monitoring tool. Even though the problem may be easy to solve, by the time the SA receives this ticket, the system may already be having problems for an extended time, disrupting its normal operations. Using the profiler, such problems can be accurately predicted and notifications can be provided to SAs earlier.

Experience with Application Problems

Application problems are the most frequently encountered category of problems (as shown in Figure 4), and also the most varied due to the myriad applications running in enterprise environments. Some application problems can be detected with periodic probes similar to the way we diagnose NFS-related problems. These can check, for example, for the liveness of application processes, whether specific application ports

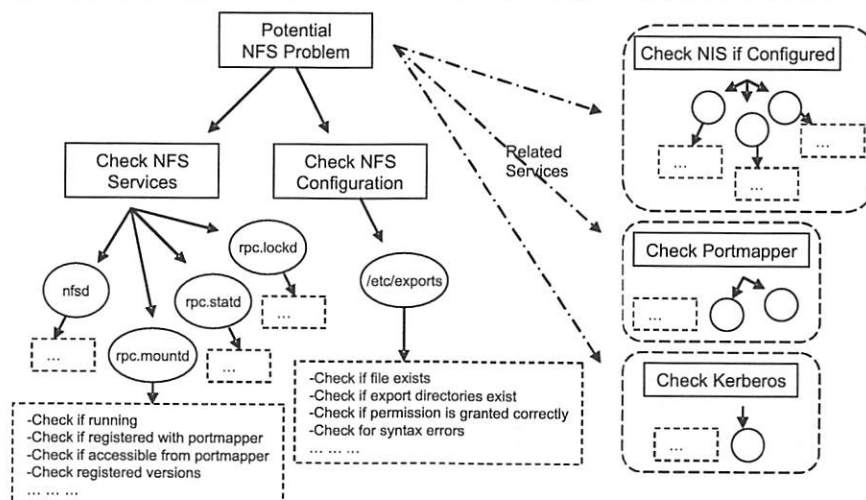


Figure 10: NFS problem determination rule tree.

are open, or for the presence of certain application files. The extensibility of PDA can be leveraged to handle other types of application problems, such as newly discovered security vulnerabilities. A community of users can devise and distribute new probes and associated rules to detect new vulnerabilities, which can shorten the system exposure time and save SAs time in evaluating whether their systems are affected.

One application for which we observed a significant number of problem tickets was the widely deployed mail server application, Sendmail. Traditionally, Sendmail configuration has been considered very complex and we expected the problems to be related primarily to setup issues or misconfiguration. However, again, most of the problems were actually caused by relatively simple issues – a congested mail queue, a dead Sendmail process, a newly discovered security vulnerability, etc. An example problem ticket related to Sendmail had the following description: *Ops received the following alerts for <hostname> advising: "Total mail queue is greater than 15000 – currently 56345. Sendmail server <hostname> is not responding to SMTP connection on port 25."* Using a periodic liveness probe and a profiler (similar to the one for filesystems described above), the SA can be notified of a pending problem well before a critical level is reached. In the current implementation, PDA incorporates rules to check some Sendmail-related parameters, including process liveness and mail queue length.

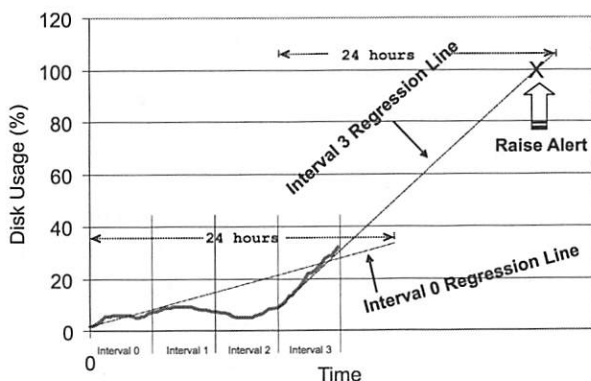


Figure 11: Using profiler to find storage capacity problems. The time axis is not drawn to scale.

Related Work

Though in certain cases, problems can be solved by fixing the underlying systems. However, a large quantity of problems addressed by SAs are not system bugs, instead, they are configuration problems, compatibility issues, usage mistakes, etc. Much of the prior work on system problem determination has centered on system monitoring. While monitoring is clearly important for PDA, it is not a primary focus of our work. We designed PDA to be able to use data collected from any monitoring tool. Our analysis of the ticket data helps us choosing which part of the system

should be monitored, but not how to implement monitoring probes. Our focus is instead on creating rules that help in diagnosing a problem once the monitoring system detects an issue.

Recently, researchers have been building problem monitoring and analysis tools using system events and system logs. Redstone, et al. [8] propose a vision of an automated problem diagnosis system by observing symptoms and matching them against problem database. Works such as PeerPressure [11], Strider [12], and others [7] [6] address misconfiguration problems in Windows systems by building and identifying signatures of normal and abnormal Windows Registry entries. Magpie [2], FDR [10], and others [13, 3] improve system management by using fine-grained system event-tracing mechanisms and analysis. We have similar goals to these works but our approach is to bring more expert knowledge in diagnosing problems and allow for a dynamically changing set of collected events.

While most problem determination systems have a fixed set of events to collect, few use online measurements as a basis for taking further actions. Rish, et al. [9] propose an approach called *active probing* for network problem determination, and Banga [1] describes a similar system on built for Network Appliance storage devices. Our system bears some resemblance to these approaches but is more generalized and has tighter link with observed problems, for example through IT problem tracking systems. In making decisions about what additional information needs to be probed, we use expert knowledge and indicators collected from real tickets as opposed to using probabilistic inference.

Our rules are conceptually similar to procedures, however, they are fundamentally different. Our rule combines knowledge and execution environment. The rule execution has intelligence to identify what is the next step and to execute the right diagnostic probes. Moreover, it is important to catch the system status when the problem just occurs. In current practice, SAs often need to reproduce a problem but the environment may have changed. So performing diagnosis right after the problem occurs not only saves time, but also could be the only way to catch the root cause.

Discussion and Ongoing Work

In our design of PDA, we were able to use problem ticket information as a guideline for the design of problem determination rules and associated probes. However, deriving rules from tickets still involves manual effort. We are investigating approaches to make this process more automated, but the varying quality of the free-text descriptions will be a continuing challenge. We are also investigating better models for the structured data which can more precisely capture problem signatures.

We use structured fields such as ticket type and cause code to categorize problem tickets and pick the

common problems as described in Section Common OS problems. Qualitatively speaking, our rules are able to address these common problems, but a more quantitative notion of "problem coverage" is required. One approach we plan to pursue to this end is to extract distinguishing attributes of a small set of tickets in the categories that are covered by our rules. The attributes can include a combination of structured fields as well as keywords from the unstructured text. Using these distinguishing attributes, we can examine a much larger set of tickets and look for matching tickets automatically to calculate the fraction of tickets that we expect to be similarly covered by the problem determination rules.

Although our experience with PDA shows promise that it can reduce time or effort for diagnosing problems, a more comprehensive study is needed to gain a sense of how much time or effort can be saved. We are making PDA available to system administrators who support a variety of customer environments in order to collect additional experiences, and ideally some quantitative data on the savings. We also expect to further validate and expand our problem determination rules through usage by SAs. Since there is no effective way to document problem resolution experience, our rule and authoring mechanism are good candidate for such a purpose.

Summary

This paper describes the Problem Determination Advisor, a tool for automating the problem determination process. Based on a study of problem tickets from a large enterprise IT support organization, we identified commonly occurring server problems and developed a set of problem determination rules to aid in their diagnosis. We implement these rules in the PDA tool using a two-level approach in which high-level system health monitors trigger lower-level diagnostic probes to collect relevant details when a problem is detected. We demonstrated the effectiveness of PDA in problem diagnosis using a number of actual problem scenarios.

Acknowledgments

We are grateful to our anonymous reviewers for their thoughtful and valuable feedback. We also want to thank our shepherd, Chad Verbowski, for his collaboration in improving the paper.

Author Biographies

Hai Huang is a Research Staff Member at IBM T. J. Watson Research Center. He worked on numerous power-management projects in the past, and his current interest is in system and application management and how to make the process more autonomous. He received his Ph.D. degree in Computer Science and Engineering from the University of Michigan. He can be reached at haih@us.ibm.com.

Raymond B. Jennings, III is an advisory engineer at the IBM T. J. Watson Research Center, Yorktown Heights, New York. He works in the area of network system software and enterprise networking. He received his BS in Electrical Engineering from Western New England College and his MS in Computer Engineering from Manhattan College. He may be reached at raymondj@us.ibm.com.

Yaoping Ruan is a Research Staff Member at IBM T. J. Watson Research Center. His research interests include system management, performance analysis and optimization, and server applications. He received his Ph.D. degree in Computer Science from Princeton University. He can be reached at yaoping.ruan@us.ibm.com.

Ramendra Sahoo belongs to Systems and Network Services Group at IBM T. J. Watson Research Center. His research interests include business process management, distributed and fault-tolerant computing, numerical and parallel algorithms and data mining. He earned his B.E. (honors) from the National Institute of Technology, Durgapur, M.S. and Ph.D. from Indian Institute of Technology, Chennai and State University of New York at Stony Brook respectively. He can be reached at rsahoo@us.ibm.com.

Sambit Sahu is a Research Staff Member at IBM T. J. Watson Research Center. His interests include network and system management, performance analysis, network measurement, and Internet services. He earned his Ph.D. in Computer Science from the University of Massachusetts. He may be reached at sambits@us.ibm.com.

Anees Shaikh manages the Systems and Network Services group at the IBM TJ Watson Research Center. His interests are in network and systems management, Internet services, and network measurement. He earned B.S. and M.S. degrees in Electrical Engineering from the University of Virginia, and a Ph.D. in Computer Science and Engineering from the University of Michigan. He may be reached at aashaikh@watson.ibm.com.

Bibliography

- [1] Banga, Gaurav, "Auto-Diagnosis of Field Problems in an Appliance Operating System," *Usenix Annual Technical Conference*, 2000.
- [2] Barham, Paul, Austin Donnelly, Rebecca Isaacs and Richard Mortier, "Using magpie For Request Extraction and Workload Modelling," *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.
- [3] Cohen, Ira, et al., "Capturing, Indexing, Clustering, and Retrieving System History," *Symposium on Operating Systems Principles*, 2005.
- [4] Rudd, Colin, "An Introductory Overview of ITIL," *IT Service Management Forum*, April, 2004, <http://www.itsmfusa.org>.

- [5] Ganapathi, Archana, Viji Ganapathi, and David Patterson, "Windows XP Kernel Crash Analysis," *20th Large Installation System Administration Conference*, 2006.
- [6] Ganapathi, Archana, Yi-Min Wang, Ni Lao, and Ji-Rong Wen, "Why PCs Are Fragile and What We Can Do About It: A Study Of Windows Registry Problems," *International Conference on Dependable Systems and Networks*, 2004.
- [7] Lao, Ni, et al., "Combining High Level Symptom Descriptions and Low Level State Information For Configuration Fault Diagnosis," *19th Large Installation System Administration Conference (LISA '04)*, Atlanta, GA, Nov., 2004.
- [8] Redstone, Joshua, Michael M. Swift and Brian N. Bershad, "Using Computers to Diagnose Computer Problems," *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2004.
- [9] Rish, Irina, et al., "Real-Time Problem Determination in Distributed Systems Using Active Probing," *IEEE/IFIP Network Operations and Management Symposium*, pp. 133-146, Seoul, Korea, April 2004.
- [10] Verbowski, Chad, et al., "Flight Data Recorder: Monitoring Persistent-state interactions To Improve Systems Management," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov., 2006.
- [11] Wang, Helen, et al., "Automatic Misconfiguration Troubleshooting With Peerpressure," *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec., 2004.
- [12] Wang, Yi-Min, et al., "Strider: A Black-Box, State-Based Approach to Change and Configuration Management and Support," *17th Large Installation System Administration Conference*, 2003.
- [13] Yuan, Chun, et al., "Automated Known Problem Diagnosis with Event Traces," *EuroSys*, 2006.

Usher: An Extensible Framework for Managing Clusters of Virtual Machines

Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker
– University of California, San Diego

ABSTRACT

Usher is a virtual machine management system designed to impose few constraints upon the computing environment under its management. Usher enables administrators to choose how their virtual machine environment will be configured and the policies under which they will be managed. The modular design of Usher allows for alternate implementations for authentication, authorization, infrastructure handling, logging, and virtual machine scheduling. The design philosophy of Usher is to provide an interface whereby users and administrators can request virtual machine operations while delegating administrative tasks for these operations to modular plugins. Usher's implementation allows for arbitrary action to be taken for nearly any event in the system. Since July 2006, Usher has been used to manage virtual clusters at two locations under very different settings, demonstrating the flexibility of Usher to meet different virtual machine management requirements.

Introduction

Usher is a cluster management system designed to substantially reduce the administrative burden of managing cluster resources while simultaneously improving the ability of users to request, control, and customize their resources and computing environment. System administrators of cluster computing environments face a number of imposing challenges. Different users within a cluster can have a wide range of computing demands, spanning general best-effort computing needs, batch scheduling systems, and complete control of dedicated resources. These resource demands vary substantially over time in response to changes in workload, user base, and failures. Furthermore, users often need to customize their operating system and application environments, substantially increasing configuration and maintenance tasks. Finally, clusters rarely operate in isolated administrative environments, and must be integrated into existing authentication, storage, network, and host address and name service infrastructure.

Usher balances these imposing requirements using a combination of abstraction and architecture. Usher provides a simple abstraction of a logical cluster of virtual machines, or virtual cluster. Usher users can create any number of virtual clusters (VCs) of arbitrary size, while Usher multiplexes individual virtual machines (VMs) on available physical machine hardware. By decoupling logical machine resources from physical machines, users can create and use machines according to their needs rather than according to assigned physical resources.

Architecturally, Usher is designed to impose few constraints upon the computing environment under its management. No two sites have identical hardware and software configurations, user and application

requirements, or service infrastructures. To facilitate its use in a wide range of environments, Usher combines a core set of interfaces that implement basic mechanisms, clients for using these mechanisms, and a framework for expressing and customizing administrative policies in extensible modules, or plugins.

The Usher core implements basic virtual cluster and machine management mechanisms, such as creating, destroying, and migrating VMs. Usher clients use this core to manipulate virtual clusters. These clients serve as interfaces to the system for users as well as for use by higher-level cluster software. For example, an Usher client called *ush* provides an interactive command shell for users to interact with the system. We have also implemented an adapter for a high-level execution management system [6], which operates as an Usher client, that creates and manipulates virtual clusters on its own behalf.

Usher supports customizable modules for two important purposes. First, these modules enable Usher to interact with broader site infrastructure, such as authentication, storage, and host address and naming services. Usher implements default behavior for common situations, e.g., newly created VMs in Usher can use a site's DHCP service to obtain addresses and domain names. Additionally, sites can customize Usher to implement more specialized policies; at UCSD, an Usher VM identity module allocates IP address ranges to VMs within the same virtual cluster.

Second, pluggable modules enable system administrators to express site-specific policies for the placement, scheduling, and use of VMs. As a result, Usher allows administrators to decide how to configure their virtual machine environments and determine the appropriate management policies. For instance, to support a general-purpose computing environment, administrators can

install an available Usher scheduling and placement plugin that performs round-robin placement of VMs across physical machines and simple rebalancing in response to the addition or removal of virtual and physical machines. With this plugin, users can dynamically add or remove VMs from VCs at any time without having to specify service level agreements (SLAs) [9, 17, 22], write configuration files [10], or obtain leases on resources [12, 14]. With live migration of VMs, Usher can dynamically and transparently adjust the mapping of virtual to physical machines to adapt to changes in load among active VMs or the working set of active VMs, exploit affinities among VMs (e.g., to enhance physical page sharing [20]), or add and remove hardware with little or no interruption.

Usher enables other powerful policies to be expressed, such as power management (reduce the number of active physical machines hosting virtual clusters), distribution (constrain virtual machines within a virtual cluster to run on separate nodes), and resource guarantees. Another installation of Usher uses its cluster to support scientific batch jobs running within virtual clusters, guarantees resources to those jobs when they run, and implements a load-balancing policy that migrates VMs in response to load spikes [13].

Usher is a fully functional system. It has been installed in cluster computing environments at UCSD and the Russian Research Center in Kurchatov, Russia. At UCSD, Usher has been in production use since January 2007. It has managed up to 142 virtual machines in 26 virtual clusters across 25 physical machines. The Usher implementation is sufficiently reliable that we are now migrating the remainder of our user base from dedicated physical machines to virtual clusters, and Usher will soon manage all 130 physical nodes in our cluster. In the rest of this paper we describe the design and implementation of Usher, as well as our experiences using it.

Related Work

Since the emergence of widespread cluster computing over a decade ago [8, 16], many cluster configuration and management systems have been developed to achieve a range of goals. These goals naturally influence individual approaches to cluster management. Early configuration and management systems, such as Galaxy [19], focus on expressive and scalable mechanisms for defining clusters for specific types of service, and physically partition cluster nodes among those types.

More recent systems target specific domains, such as Internet services, computational grids, and experimental testbeds, that have strict workload or resource allocation requirements. These systems support services that express explicit resource requirements, typically in some form of service level agreement (SLA). Services provide their requirements as input to the system, and the system allocates its resources

among services while satisfying the constraints of the SLA requirements.

For example, Océano provides a computing utility for e-commerce [9]. Services formally state their workload performance requirements (e.g., response time), and Océano dynamically allocates physical servers in response to changing workload conditions to satisfy such requirements. Rocks and Rolls provide scalable and customizable configuration for computational grids [11, 18], and Cluster-on-Demand (COD) performs resource allocation for computing utilities and computational grid services [12]. COD implements a virtual cluster abstraction, where a virtual cluster is a disjoint set of physical servers specifically configured to the requirements of a particular service, such as a local site component of a larger wide-area computational grid. Services specify and request resources to a site manager and COD leases those resources to them. Finally, Emulab provides a shared network testbed in which users specify experiments [21]. An experiment specifies network topologies and characteristics as well as node software configurations, and Emulab dedicates, isolates, and configures testbed resources for the duration of the experiment.

The recent rise in virtual machine monitor (VMM) popularity has naturally led to systems for configuring and managing virtual machines. For computational grid systems, for example, Shirako extends Cluster-on-Demand by incorporating virtual machines to further improve system resource multiplexing while satisfying explicit service requirements [14], and VIO-LIN supports both intra- and inter-domain migration to satisfy specified resource utilization limits [17]. Sandpiper develops policies for detecting and reacting to hotspots in virtual cluster systems while satisfying application SLAs [22], including determining when and where to migrate virtual machines, although again under the constraints of meeting the stringent SLA requirements of a data center.

On the other hand, Usher provides a framework that allows system administrators to express site-specific policies depending upon their needs and goals. By default, the Usher core provides, in essence, a general-purpose, best-effort computing environment. It imposes no restrictions on the number and kind of virtual clusters and machines, and performs simple load balancing across physical machines. We believe this usage model is important because it is widely applicable and natural to use. Requiring users to explicitly specify their resource requirements for their needs, for example, can be awkward and challenging since users often do not know when or for how long they will need resources. Further, allocating and reserving resources can limit resource utilization; guaranteed resources that go idle cannot be used for other purposes. However, sites can specify more elaborate policies in Usher for controlling the placement, scheduling, and migration of VMs if desired. Such policies can range

from batch schedulers to allocation of dedicated physical resources.

In terms of configuration, Usher shares many of the motivations that inspired the Manage Large Networks (MLN) tool [10]. The goal of MLN is to enable administrators and users to take advantage of virtualization while easing administrator burden. Administrators can use MLN to configure and manage virtual machines and clusters (*distributed projects*), and it supports multiple virtualization platforms (Xen and User-Mode Linux). MLN, however, requires administrators to express a number of static configuration decisions through configuration files (e.g., physical host binding, number of virtual hosts), and supports only coarse granularity dynamic reallocation (manually by the administrator). Usher configuration is interactive and dynamic, enables users to create and manage their virtual clusters without administrative intervention, and enables a site to globally manage all VMs according to cluster-wide policies.

XenEnterprise [5] from XenSource and VirtualCenter [4] from VMware are commercial products for managing virtual machines on cluster hardware from the respective companies. XenEnterprise provides a graphical administration console, Virtual Data Center, for creating, managing, and monitoring Xen virtual machines. VirtualCenter monitors and manages VMware virtual machines on a cluster as a data center, supporting VM restart when nodes fail and dynamic load balancing through live VM migration. Both list interfaces for external control, although it is not clear whether administrators can implement arbitrary plugins and policies for integrating the systems into existing infrastructure, or controlling VMs in response to arbitrary events in the system. In this regard, VMWare's Infrastructure Management SDK provides functionality similar to that provided by the Usher client API. However, this SDK does not provide the tight integration with VMWare's centralized management system that plugins do for the Usher system. Also, of course, these are all tied to managing a single VM product, whereas Usher is designed to interface with any virtualization platform that exports a standard administrative interface.

System Architecture

This section describes the architecture of Usher. We start by briefly summarizing the goals guiding our design, and then present a high-level overview of the system. We then describe the purpose and operation of each of the various system components, and how they interact with each other to accomplish their tasks. We end with a discussion of how the Usher system accommodates software and hardware failures.

Design Goals

As mentioned, no two sites have identical hardware and software configurations, user and application requirements, or service infrastructures. As a result,

we designed Usher as a flexible platform for constructing virtual machine management installations customized to the needs of a particular site.

To accomplish this goal, we had two design objectives for Usher. First, Usher maintains a clean separation between policy and mechanism. The Usher core provides a minimal set of mechanisms essential for virtual machine management. For instance, the Usher core has mechanisms for placing and migrating virtual machines, while administrators can install site-specific policy modules that govern where and when VMs are placed.

Second, Usher is designed for extensibility. The Usher core provides three ways to extend functionality, as illustrated in Figure 1. First, Usher provides a set of *hooks* to integrate with existing infrastructure. For instance, while Usher provides a reference implementation for use with the Xen VMM, it is straightforward to write stubs for other virtualization platforms. Second, developers can use a *Plugin API* to enhance Usher functionality. For example, plugins can provide database functionality for persistently storing system state using a file-backed database, or provide authentication backed by local UNIX passwords. Third, Usher provides a *Client API* for integrating with user interfaces and third-party tools, such as the Usher command-line shell and the Plush execution management system (discussed in the Applications subsection of the Implementation section).

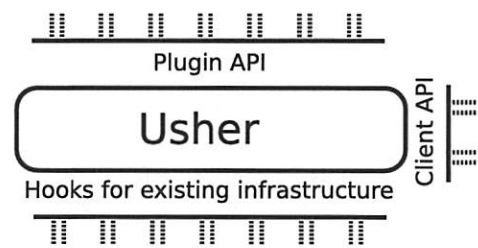


Figure 1: Usher interfaces.

Usher Overview

A running Usher system consists of three main components: local node managers (LNMs), a centralized controller, and clients. A client consists of an application that utilizes the Usher client library to send virtual machine management requests to the controller. We have written a few applications that import the Usher client library for managing virtual machines (a shell and XML-RPC server) with more under development (a web frontend and command line suite).

Figure 2 depicts the core components of an Usher installation. One LNM runs on each physical node and interacts directly with the VMM to perform management operations such as creating, deleting, and migrating VMs on behalf of the controller. The local node managers also collect resource usage data from the VMMs and monitor local events. LNMs report resource usage updates and events back to the controller for use by plugins and clients.

The controller is the central component of the Usher system. It receives authenticated requests from clients and issues authorized commands to the LNMs. It also communicates with the LNMs to collect usage data and manage virtual machines running on each physical node. The controller provides event notification to clients and plugins registered to receive notification for a particular event (e.g., a VM has started, been destroyed, or changed state). Plugin modules can perform a wide range of tasks, such as maintaining persistent system-wide state information, performing DDNS updates, or doing external environment preparation and cleanup.

The client library provides an API for applications to communicate with the Usher controller. Essentially, clients submit requests to the controller when they need to manipulate their VMs or request additional VMs. The controller can grant or deny these requests as its operational policy dictates. One purpose of clients is to serve as the user interface to the system, and users use clients to manage their VMs and monitor system state. More generally, arbitrary applications can use the client library to register callbacks for events of interest in the Usher system.

Typically, a few services also support a running Usher system. Depending upon the functionality desired and the infrastructure provided by a particular site, these services might include a combination of the following: a database server for maintaining state information or logging, a NAS server to serve VM filesystems, an authentication server to provide authentication for Usher and VMs created by Usher, a DHCP server to manage IP addresses, and a DNS server for name resolution of all Usher created VMs. Note that an administrator may configure Usher to use any set of support services desired, not necessarily the preceding list.

Usher Components

As noted earlier, Usher consists of three main components, local node managers on each node, a central controller, and Usher clients.

Local Node Managers

The local node managers (LNMs) operate closest to the hardware. As shown in Figure 2, LNMs run as servers on each physical node in the Usher system. The LNMs have three major duties: i) to provide a remote API to the controller for managing local VMs, ii) to collect and periodically upload local resource usage data to the controller, and iii) to report local events to the controller.

Each LNM presents a remote API to the controller for manipulating VMs on its node. Upon invoking an API method, the LNM translates the operation into the equivalent operation of the VM management API exposed by the VMM running on the node. Note that all LNM API methods are asynchronous so that the controller does not block waiting for the VMM

operation to complete. We emphasize that this architecture abstracts VMM-specific implementations – the controller is oblivious to the specific VMMs running on the physical nodes as long as the LNM provides the remote API implementation. As a result, although our implementation currently uses the Xen VMM, Usher can target other virtualization platforms. Further, Usher is capable of managing VMs running any operating system supported by the VMMs under its management.

As the Usher system runs, VM and VMM resource usage fluctuates considerably. The local node manager on each node monitors these fluctuations and reports them back to the controller. It reports resource usage of CPU utilization, network receive and transmit loads, disk I/O activity, and memory usage in 1, 5, and 15-minute averages.

In addition to changes in resource usage, VM state changes sometimes occur unexpectedly. VMs can crash or even unexpectedly appear or disappear from the system. Detecting these and other related events requires both careful monitoring by the local node managers as well as VMM support for internal event notification. Administrators can set a tunable parameter for how often the LNM scans for missing VMs or unexpected VMs. The LNM will register callbacks with the VMM platform for other events, such as VM crashes; if the VMM does not support such callbacks, LNM will periodically scan to detect these events.

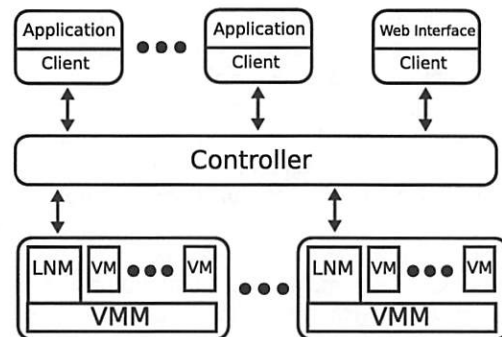


Figure 2: Usher components.

Usher Controller

The controller is the center of the Usher system. It can either be bootstrapped into a VM running in the system, or run on a separate server. The controller provides the following:

- User authentication
- VM operation request API
- Global state maintenance
- Consolidation of LNM monitoring data
- Event notification

User authentication: Usher uses SSL-encrypted user authentication. All users of the Usher system must authenticate before making requests of the system. Administrators are free to use any of the included authentication modules for use with various authentication

backends (e.g., LDAP), or implement their own. An administrator can register multiple authentication modules, and Usher will query each in turn. This support is useful, for instance, to provide local password authentication if LDAP or NIS authentication fails. After receiving a user's credentials, the controller checks them against the active authentication module chain. If one succeeds before reaching the end of the chain, the user is authenticated. Otherwise, authentication fails and the user must retry.

VM operation request API: A key component of the controller is the remote API for Usher clients. This API is the gateway into the system for VM management requests (via RPC) from connecting clients. Typically, the controller invokes an authorization plugin to verify that the authenticated user can perform the operation before proceeding. The controller may also invoke other plugins to do preprocessing such as checking resource availability and making placement decisions at this point. Usher calls any plugin modules registered to receive notifications for a particular request once the controller receives such a request.

Usher delegates authorization to plugin modules so that administrators are free to implement any policy or policies they wish and stack and swap modules as the system runs. In addition, an administrator can configure the monitoring infrastructure to automatically swap or add policies as the system runs based upon current system load, time of day, etc. In its simplest form, an authorization policy module openly allows users to create and manipulate their VMs as they desire or view the global state of the system. More restrictive policies may limit the number of VMs a user can start, prohibit or limit migration, or restrict what information the system returns upon user query.

Once a request has successfully traversed the authorization and preprocessing steps, the controller executes it by invoking asynchronous RPCs to each LNM involved. As described above, it is up to any plugin policy modules to authorize and check resource availability prior to this point. Depending upon the running policy, the authorization and preprocessing steps may alter a user request before the controller executes it. For example, the policy may be to simply "do the best I can" to honor a request when it arrives. If a user requests more VMs than allowed, this policy will simply start as many VMs as are allowed for this user, and report back to the client what action was taken. Finally, if insufficient resources are available to satisfy an authorized and preprocessed request, the controller will attempt to fulfill the request until resources are exhausted.

Global state maintenance: The controller maintains a few lists which constitute the global state of the system. These lists link objects encapsulating state information for running VMs, running VMMs, and instantiated virtual clusters (VCs). A virtual cluster in Usher can contain an arbitrary set of VMs, and administrators are free to define VCs in any way suitable to their computing environment.

In addition to the above lists, the controller maintains three other lists of VMs: *lost*, *missing*, and *unmanaged* VMs. The subtle distinction between lost and missing is that lost VMs are a result of an LNM or VMM failure (the controller is unable to make this distinction), whereas a missing VM is a result of an unexpected VM disappearance as reported by the LNM where the VM was last seen running. A missing VM can be the result of an unexpected VMM error (e.g., we have encountered this case upon a VMM error on migration). Unmanaged VMs are typically a result of an administrator manually starting a VM on a VMM being managed by Usher; Usher is aware of the VM, but is not itself managing it. The list of unmanaged VMs aids resource usage reporting so that Usher has a complete picture of all VMs running on its nodes.

Having the controller maintain system state removes the need for it to query all LNMs in the system for every VM management operation and state query. However, the controller does have to synchronize with the rest of system and we discuss synchronization further in the Component Interaction subsection below.

Consolidation of LNM monitoring data: Proper state maintenance relies upon system monitoring. The controller is responsible for consolidating monitoring data sent by the local node managers into a format accessible by the rest of the system. Clients use this data to describe the state of the system to users, and plugins use this data to make policy decisions. For example, plugin modules may use this data to restrict user resource requests based on the current system load, or make VM scheduling decisions to determine where VMs should run.

Event notification: Usher often needs to alert clients and plugin modules when various events in the system occur. Events typically fall into one of three categories:

- VM operation requests
- VM state changes
- Errors and unexpected events

Clients automatically receive notices of state changes of their virtual machines. Clients are free to take any action desired upon notification, and can safely ignore them. Plugin modules, however, must explicitly register with the controller to receive event notifications. Plugins can register for any type of event in the system. For example, a plugin may wish to receive notice of VM operation requests for preprocessing, or error and VM state change events for reporting and cleanup.

Clients and the Client API

Applications use the Usher client API to interact with the Usher controller. This API provides methods for requesting or manipulating VMs and performing state queries. We refer to any application importing this API as a client.

The client API provides the mechanism for clients to securely authenticate and connect to the Usher controller. Once connected, an application may call any of the methods provided by the API. All methods are asynchronous, event-based calls to the controller (see the Implementation section below). As mentioned above, connected clients also receive notifications from the controller for state changes to any of their VMs. Client applications can have their own callbacks invoked for these notifications.

Component Interaction

Having described each of the Usher components individually, we now describe how they interact in more detail. We first discuss how the controller and LNMs interact, and then describe how the controller and clients interact. Note that clients never directly communicate with LNMs; in effect, the controller “proxies” all interactions between clients and LNMs.

Controller and LNM Interaction

When an LNM starts or receives a controller recovery notice, it connects to the controller specified in its configuration file. The controller authenticates all connections from LNMs, and encrypts the connection for privacy. Upon connection to the controller, the LNM passes a capability to the controller for access to its VM management API.

Using the capability returned by the LNM, the controller first requests information about the hardware configuration and a list of currently running virtual machines on the new node. The controller adds this information to its lists of running VMs and VMMs in the system. It then uses the capability to assume management of the VMs running on the LNM's node.

The controller also returns a capability back to the LNM. The LNM uses this capability for both event notification and periodic reporting of resource usage back to the controller.

When the controller discovers that a new node already has running VMs (e.g., because the node's LNM failed and restarted), it first determines if it should assume management of any of these newly discovered VMs. The controller makes this determination based solely upon the name of the VM. If the VM name ends with the domain name specified in the controller's configuration file, then the controller assumes it should manage this VM. Any VMs which it should not manage are placed on the *unmanaged* list discussed above. For any VMs which the controller should manage, the controller creates a VM object instance and places this object on its running VMs list. These instances are sent to the LNMs where the VMs are running and cached there. Whenever an LNM sees that a cached VM object is inconsistent with the corresponding VM running there (e.g., the state of the VM changed), it alerts the controller of this event. The controller then updates the cached object on the LNM.

In this way, the update serves as an acknowledgment and the LNM knows that the controller received notice of the event.

Similarly, the controller sends VM object instances for newly created VMs to an LNM before the VM is actually started there. Upon successful return from a start command, the controller updates the VMs cached object state on the LNM. Subsequently, the LNM assumes the responsibility for monitoring and reporting any unexpected state changes back to the controller.

Controller and Client Interaction

Clients to the Usher system communicate with the controller. Before a client can make any requests, it must authenticate with the controller. If authentication succeeds, the controller returns a capability to the client for invoking its remote API methods. Clients use this API to manipulate VMs.

Similar to the local node managers, clients receive cached object instances corresponding to their VMs from the controller upon connection. If desired, clients can filter this list of VMs based upon virtual cluster grouping to limit network traffic. The purpose of the cached objects at the client is twofold. First, they provide a convenient mechanism by which clients can receive notification of events affecting their VMs, since the controller sends updates to each cached VM object when the actual VM is modified. Second, the cached VM objects provide state information to the clients when they request VM operations. With this organization, clients do not have to query the controller about the global state of the system before actually submitting a valid request. For example, a client should not request migration of a non-existent VM, or try to destroy a VM which it does not own. The client library is designed to check for these kinds of conditions before submitting a request. Note that the controller is capable of handling errant requests; this scheme simply offloads request filtering to the client.

The controller is the authority on the global state of the system. When the controller performs an action, it does so based on what it believes is the current global state. The cached state at the client reflects the controller's global view. For this reason, even if the controller is in error, its state is typically used by clients for making resource requests. The controller must be capable of recovering from errors due to inconsistencies between its own view of the global state of the system and the actual global state. These inconsistencies are typically transient (e.g., a late event notification from an LNM), in which case the controller may log an error and return an error message to the client.

Failures

As the Usher system runs, it is possible for the controller or any of the local node managers to become unavailable. This situation could be the result of hardware failure, operating system failure, or the server itself

failing. Usher has been designed to handle these failures gracefully.

In the event of a controller failure, the LNMs will start a listening server for a recovery announcement sent by the controller. When the controller restarts, it sends a recovery message to all previously known LNMs. When the LNMs receive this announcement, they reconnect to the controller. As mentioned in the Controller and LNM Interaction section above, when an LNM connects to the controller, it passes information about its physical parameters and locally running VMs. With this information from all connecting LNMs, the controller recreates the global state of the system. With this design, Usher only requires persistent storage of the list of previously known LNMs rather than the entire state of the system to restore system state upon controller crash or failure.

Since the controller does not keep persistent information about which clients were known to be connected before a failure, it cannot notify clients when it restarts. Instead, clients connected to a controller which fails will attempt to reconnect with timeouts following an exponential backoff. Once reconnected, clients flush their list of cached VMs and receive a new list from the controller.

The controller detects local node manager failures upon disconnect or TCP timeout. When this situation occurs, the controller changes the state of all VMs known to be running on the node with the failed LNM to *lost*. It makes no out of band attempts to determine if lost VMs are still running or if VMMs on which LNMs have failed are still running. The controller simply logs an error, and relies upon the Usher administrator or a recovery plugin to investigate the cause of the error.

Implementation

In this section we describe the implementation of Usher, including the interfaces that each component supports and the plugins and applications currently implemented for use with the system.

Component	LoC
LNM (w/ Xen hooks)	907
Controller	1703
Client API	750
Utilities	633
Ush	1099

Table 1: Code size of individual components.

The main Usher components are written in Python [2]. In addition, Usher makes use of the Twisted network programming framework [3]. Twisted provides convenient mechanisms for implementing event based servers, asynchronous remote procedure calls, and remote object synchronization. Table 1 shows source code line counts for the main Usher components, for

total of 3993 lines of code. Also included is the line count for the *ush* application (over 400 of which is simply online documentation).

Local Node Managers

Local Node Managers export the remote API shown in Table 2 to the controller. This API is made available to the controller via a capability passed to the controller when an LNM connects to it.

Method Name	Description
get_details(vm name)	get VM state information
get_status(vm name)	get VM resource usage statistics
receive(vm instance)	receive new cached VM object
start(vm name)	start cached VM
op_on(operation, vm name)	operate on existing VM
migrate(vm name, lnm name)	migrate VM to LNM
get_node_info()	get node physical characteristics
get_node_status()	get node dynamic and resource usage info

Table 2: Local node manager remote API.

This API includes methods to query for VM state information and VM resource usage details using the *get_details* and *get_status* methods, respectively. State information includes run state, memory allocation, IP and MAC addresses, the node on which VM is running, VM owner, etc. Resource usage includes 1, 5, and 15-minute utilizations of the various hardware resources.

The *receive* method creates a cached copy of a VM object on an LNM. An LNM receives the cached copy when it connects to the controller. It compares the state of the VM object with the actual state of the virtual machine. If the states differ the LNM notifies the controller, which updates the LNM's cached copy of the VM as an acknowledgment that it received the state change notice.

In addition, the cached copy of a VM at its LNM contains methods for manipulating the VM it represents. When a VM manipulation method exposed by the LNM's API is invoked (one of *start*, *op_on*, or *migrate*), the method calls the corresponding method of the cached VM object to perform the operation. This structure provides a convenient way to organize VM operations. To manipulate a VM, a developer simply calls the appropriate method of the cached VM object. Note that the controller must still update the state of its VM object as an acknowledgment that the controller knows the operation was successful.

Most operations on an existing VM are encapsulated in the `op_on` function, and have similar signatures. Table 3 shows the list of valid operations to the `op_on` method.

Operation	Description
pause	pause VM execution, keeping memory image resident
resume	resume execution of a paused VM
shutdown	nicely halt a VM
reboot	shutdown and restart VM
hibernate	save VM's memory image to persistent storage
restore	restore hibernated VM to run state
destroy	hard shutdown a VM
cycle	destroy and restart a VM

Table 3: Operations supported by the `op_on` method.

All VM operations invoke a corresponding operation in the VMM's administration API. Though Usher currently only manages Xen VMs, it is designed to be VMM-agnostic. An installation must provide an implementation of Usher's VMM interface to support new virtual machine managers.

The LNM's remote API exposes a few methods that do not operate on VMs. The `get_node_info` method returns hardware characteristics of the physical machine. The controller calls this method when an LNM connects. The `get_node_status` method is similar to the `get_status` method. Additionally, it reports the number of VMs running on the VMM and the amount of free memory on the node.

Usher Controller

The remote API exported by the controller to connecting clients closely resembles the interface exported by LNMs to the controller. Table 4 lists the methods exported by the controller to Usher clients. This API is made available to clients via a capability passed upon successful authentication with the controller.

Note that most of these methods operate on lists of VMs, rather than single VMs expected by the LNM API methods. Since Usher was designed to manage clusters, the common case is to invoke these methods on lists of VMs rather than on a single VM at a time. This convention saves significant call overhead when dealing with large lists of VMs.

The `start` and `migrate` methods both take a list of LNMs. For `start`, the list specifies the LNMs on which the VMs should be started. An empty list indicates that the VMs can be started anywhere. Recall that this parameter is simply a suggestion to the controller. Policies installed in the controller dictate whether or not the controller will honor the suggestion. Likewise, the LNM list passed to the `migrate` method is simply a

suggestion to the controller as to where to migrate the VMs. The controller can choose to ignore this suggestion or ignore the migrate request altogether based upon the policies installed.

The operations supported by the `op_on` method in the controller API are the same as those to the `op_on` method of the remote LNM API (Table 3).

Method Name	Description
<code>list(vm list, status)</code>	list state and resource usage information for VMs
<code>list_lnm(lnm list, status)</code>	list LNMs and resource usage information for VMMs
<code>start(vm list, lnm list)</code>	start list of VMs on LNMs
<code>op_on(operation, vm list)</code>	operate on existing VMs
<code>migrate(vm list, lnm list)</code>	migrate VMs to LNMs

Table 4: Controller remote API for use by Usher clients.

Client API

The client API closely mirrors that of the controller. An important difference between these two APIs, though, is that the client API signatures contain many additional parameters to aid in working with large sets of VMs. These additional parameters allow users to operate on arbitrary sets of VMs and virtual clusters in a single method call. The API supports specifying VM sets as regular expressions, explicit lists, or ranges (when VM names contain numbers). The client API also allows users to specify source and destination LNMs using regular expressions or explicit lists.

Another difference between the client and controller APIs is that the client API expands the `op_on` method into methods for each type of operation. Explicitly enumerating the operations as individual methods avoids confusing application writers unfamiliar with the `op_on` method. These methods simply wrap the call to the `op_on` method, which is still available for those wishing to call it directly.

Finally, the client API contains `connect` and `reconnect` methods. These methods contact and authenticate with the controller via SSL. They also start the client's event loop to handle cached object updates and results from asynchronous remote method calls. The `reconnect` method is merely a convenience method to avoid having to pass credentials to the API if a reconnect is required after having been successfully connected. This method can be used by a reconnecting application upon an unexpected disconnect.

Configuration Files

All Usher components are capable of reading configuration data from text files. All valid configuration parameters, their type, and default values are

specified in the code for each component. When each component starts, it first parses its configuration files (if found). The configuration system tries to read in values from the following locations (in order): a default location in the host filesystem, a default location in the user's home directory, and finally a file indicated by an environment variable. This search ordering enables users to override default values easily. Values read in later configuration files replace values specified in a previously read file.

Plugins

Plugins are separate add-on modules which can be registered to receive notification of nearly any event in the system. Plugins live in a special directory (aptly named "plugins") of the Usher source tree. Usher also looks in a configurable location for third-party/user plugins. Any plugins found are automatically sourced and added to a list of *available* plugins. To register a plugin, the controller provides an additional API call `register_plugin(plugin name, event, configuration file)`. Each plugin is required to provide a method named `entry_point` to be called when an event fires for which it is registered. It is possible to add a single plugin to multiple event handler chains. Note that the `register_plugin` method can be called from anywhere in the Usher code.

By default, plugins for each event are simply called in the order in which they are registered. Therefore careful consideration must be given to ordering while registering plugins. A plugin's configuration object can optionally take an *order* parameter that governs the order in which plugins are called on the event's callback list. The plugin API also provides a converse `unregister_plugin` call to change event handling at runtime.

Plugins can be as simple or complex as necessary. Since the controller invokes plugin callback chains asynchronously, complex plugins should not interfere with the responsiveness of the Usher system (i.e., the main controller event loop will not block waiting for a plugin to finish its task).

Policies in an Usher installation are implemented as plugins. As an example, an administrator may have strict policies regarding startup and migration of virtual machines. To enforce these policies, a plugin (or plugins) is written to authorize start and migrate requests. This plugin gets registered for the `start_request` and `migrate_request` events, either manually using the controllers `register_plugin` command, or by specifying these registrations in the controller's configuration file. Once registered, subsequent start and migrate requests are passed to the plugin (in the form of a Request object) for authorization. At this point, the plugin can approve, approve with modification, or simply reject the request. Once this is done, the request is passed on to any other plugins registered on the `start_request` or `migrate_request` event lists with a higher order attribute.

Besides authorization policies, one can imagine policies for VM operation and placement. For example, initial VM placement, VM scheduling (i.e., dynamic migration based on load or optimizing a utility function), or reservations. A policy plugin for initial placement would be registered for the `start_request` event (probably with a higher order attribute than the startup authorization policy discussed above so that it is called later in the plugin callback chain). Some simple policies such a plugin might support are round-robin and least-loaded. Scheduling and reservation plugins could be registered with a timer to be fired periodically to evaluate the state of the system and make decisions about where VMs should be migrated and which VMs might have an expired reservation, respectively.

As a concrete example of plugin usage in Usher, we now discuss plugins implemented for use by the UCSD Usher installation, and outline the sequence of events for a scenario of starting a set of VMs. Detailed discussion about these plugins is deferred to the UCSD SysNet subsection of the Usher Deployments section.

The UCSD installation uses the following plugins: an SQL database plugin for logging, mirroring global system state, and IP address management; an LDAP plugin for user authentication for both Usher and VMs created by Usher; a filesystem plugin for preparing writable VM filesystems; a DNS plugin for modifying DNS entries for VMs managed by Usher; and a default placement plugin to determine where VMs should be started. We are developing additional modules for VM scheduling as part of ongoing research.

All plugins for the UCSD installation are written in Python. Table 5 contains line counts for these plugins. Overall, the UCSD plugins total 1406 lines of code.

Plugin	LoC
Database	260
LDAP	870
Filesystem	54
DNS	90
Placement	132

Table 5: Code size of UCSD plugins.

When a request to start a list of VMs arrives, the controller calls the modules registered for the "start request" event. The placement and database modules are in the callback list for this event. The placement module first determines which VMs to start based on available resources and user limits, then determines where each of the allowed VMs will start. The database module receives the modified request list, logs the request, then reserves IP addresses for each of the new VMs.

The controller generates a separate VM start command for each VM in the start list. Prior to invoking the start command, the controller triggers a “register VM” event. The database, DNS, and file system plugin modules are registered for this event. The database module adds the VM to a “VMs” table to mirror the fact that this is now a VM included in the controller’s view of the global state. The DNS plugin simply sends a DDNS update to add an entry for this VM in our DNS server. The filesystem module prepares mount points on an NFS server for use by each VM.

Finally, upon return from each start command, a “start complete” event fires. The database module registers to receive this event. The database module checks the result of the command, logs this to the database, then either marks the corresponding IP address as used (upon success) or available (upon failure). Note that the database module does not change the state of the VM in the VMs table until receiving a “state changed” event for this VM (which originates at the LNM).

Sites can install or customize plugins as necessary. The Usher system supports taking arbitrary input and passing it to plugins on request events. For example, a request to start a VM may include information about which OS to boot, which filesystem to mount, etc. Plugin authors can use this mechanism to completely customize VMs at startup.

Applications

We have written two applications using the client API, a shell named *Ush* and an XML-RPC server named *plusher*, and are developing two other applications, a Web interface and a command-line suite. The Web interface will provide a convenient point and click interface accessible from any computer with a web browser. The command line suite will facilitate writing shell scripts to manage virtual machines.

Ush Client

The Usher shell *Ush* provides an interactive command-line interface to the API exported by the Usher controller. *Ush* provides persistent command line history and comes with extensive online help for each command. If provided, *Ush* can read connection details and other startup parameters from a configuration file. *Ush* is currently the most mature and preferred interface for interacting with the Usher system.

As an example of using the Usher system, we describe a sample *Ush* session from the UCSD Usher installation, along with a step-by-step description of actions taken by the core components to perform each request. In this example, user “mmcnett” requests ten VMs. Figure 3 contains a snapshot of *Ush* upon completion of the start command.

First a user connects to the Usher controller by running the “connect” command. In connecting, the controller receives the user’s credentials and checks them against the LDAP database. Once authentication

succeeds, the controller returns a capability for its remote API and all of user mmcnett’s VMs. The somewhat unusual output “<Command 0 result pending...>” reflects the fact that all client calls to the controller are asynchronous. When “connect” returns, *Ush* responds with the “Command 0 result:” message followed by the actual result “Connected”.

Upon connecting *Ush* saves the capability and cached VM instances sent by the controller. Once connected, the user runs the “list” command to view his currently running VMs. Since the client already has cached instances of user mmcnett’s VMs, the list command does not invoke any remote procedures. Consequently, *Ush* responds immediately indicating that user mmcnett already has two VMs running.

The user then requests the start of ten VMs in the “sneetch” cluster. In this case, the -n argument specifies the name of a cluster, and the -c argument specifies how many VMs to start in this cluster. When the controller receives this request, it first calls on the authorization and database modules to authorize the request and reserve IP addresses for the VMs to be started. Next, the controller calls the initial placement plugin to map where the authorized VMs should be started. The controller calls the start method of the remote LNM API at each new VM’s LNM. The LNMs call the corresponding method of the VMM administration API to start each VM. Upon successful return of all of these remote method calls, the controller responds to the client that the ten VMs were started in two seconds and provides information about where each VM was started. After completing their boot sequence, user mmcnett can ssh into any of his new VMs by name.

```

mmcnett@5:~ (on 5.sysnet.usher.ucsdys.net)
Usher Shell 0.2
Type '?' or 'help' for help
Use: help <command> for command specific help
ush> connect
Password:
<Command 0 result pending...>
Command 0 result:
Connected
mmcnett@ush> list
2 VMs:
=====
horton.mmcnett.usher.ucsdys.net    run    vm43.usher.ucsdys.net
usher.mmcnett.usher.ucsdys.net    run    vm52.usher.ucsdys.net
mmcnett@ush> start -n sneetch -c 10
<Command 1 result pending...>
Command 1 result:
Controller started 10 VMs in 2 seconds:
1.sneetch.mmcnett.usher.ucsdys.net started on vm43.usher.ucsdys.net
10.sneetch.mmcnett.usher.ucsdys.net started on vm44.usher.ucsdys.net
2.sneetch.mmcnett.usher.ucsdys.net started on vm47.usher.ucsdys.net
3.sneetch.mmcnett.usher.ucsdys.net started on vm48.usher.ucsdys.net
4.sneetch.mmcnett.usher.ucsdys.net started on vm49.usher.ucsdys.net
5.sneetch.mmcnett.usher.ucsdys.net started on vm50.usher.ucsdys.net
6.sneetch.mmcnett.usher.ucsdys.net started on vm51.usher.ucsdys.net
7.sneetch.mmcnett.usher.ucsdys.net started on vm52.usher.ucsdys.net
8.sneetch.mmcnett.usher.ucsdys.net started on vm53.usher.ucsdys.net
9.sneetch.mmcnett.usher.ucsdys.net started on vm54.usher.ucsdys.net
mmcnett@ush>

```

Figure 3: Ush

Plusher

Plush [7] is an extensible execution management system for large-scale distributed systems, and *plusher* is an XML-RPC server that integrates Plush with Usher. Plush users describe batch experiments or computations in a domain-specific language. Plush uses this

input to map resource requirements to physical resources, bind a set of matching physical resources to the experiment, set up the execution environment, and finally execute, monitor and control the experiment.

Since Usher is essentially a service provider for the virtual machine “resource”, it was natural to integrate it with Plush. This integration enables users to request virtual machines (instead of physical machines) for running their experiments using a familiar interface.

Developing *plusher* was straightforward. Plush already exports a simple control interface through XML-RPC to integrate with resource providers. Plush requires providers to implement a small number of up-calls and down-calls. Up-calls allow resource providers to notify Plush of asynchronous events. For example, using down-calls Plush requests resources asynchronously so that it does not have to wait for resource allocation to complete before continuing. When the provider finishes allocating resources, it notifies Plush using an up-call.

To integrate Plush and Usher in *plusher*, we only needed to implement stubs for this XML-RPC interface in Usher. The XML-RPC stub uses the Client API to talk to the Usher controller. The XML-RPC stub acts as a proxy for authentication – it relays the authentication information (provided by users to Plush) to the controller before proceeding. When the requested virtual machines have been created, *plusher* returns a list of IP addresses to Plush. If the request fails, it returns an appropriate error message.

Usher Deployments

Next we describe two deployments of Usher that are in production use at different sites. The first deployment is for the UCSD CSE Systems and Networking research group, and the second deployment is at the Russian Research Center, Kurchatov Institute (RRC-KI). The two sites have very different usage models and computing environments. In describing these deployments, our goal is to illustrate the flexibility of Usher to meet different virtual machine management requirements and to concretely demonstrate how sites can extend Usher to achieve complex management goals. Usher does not force one to setup or manage their infrastructure as done by either of these two installations.

UCSD SysNet

The UCSD CSE Systems and Networking (SysNet) research group has been using Usher experimentally since June 2006 and for production since January 2007. The group consists of nine faculty, 50 graduate students, and a handful of research staff and undergraduate student researchers. The group has a strong focus on experimental networking and distributed systems research, and most projects require large numbers of machines in their research. As a result, the demand for

machines far exceeds the supply of physical machines, and juggling physical machine allocations never satisfies all parties. However, for most of their lifetimes, virtual machines can satisfy the needs of nearly all projects: resource utilization is bursty with very low averages (1% or less), an ideal situation for multiplexing; virtualization overhead is an acceptable trade-off to the benefits Usher provides; and users have complete control over their clusters of virtual machines, and can fully customize their machine environments. Usher can also isolate machines, or even remove them from virtualization use, for particular circumstances (e.g., obtaining final experimental results for a paper deadline) and simply place them back under Usher management when the deadline passes.

At the time of this writing, we have staged 25 physical machines from our hardware cluster into Usher. On those machines, Usher has multiplexed up to 142 virtual machines in 26 virtual clusters, with an average of 63 VMs active at any given time. Our Usher controller runs on a Dell PowerEdge 1750 with a 2.8 GHz processor and 2 GB of physical memory. This system easily handles our workload. Although load is mostly dictated by plugin complexity, using the plugins discussed below, the Usher controller consumes less than 1 percent CPU on average (managing 75 virtual machines) with a memory footprint of approximately 20 MB. The Usher implementation is sufficiently reliable that we are now migrating the remainder of our user base from dedicated physical machines to virtual clusters, and Usher will soon manage all 130 physical nodes in our cluster.

Usher Usage

The straightforward ability to both easily create arbitrary numbers of virtual machines as well as destroy them has proved to be very useful, and the SysNet group has used this capability in a variety of ways. As expected, this ability has greatly eased demand for physical machines within the research group. Projects simply create VMs as necessary. Usher has also been used to create clusters of virtual machines for students in a networking course; each student can create a cluster on demand to experiment with a distributed protocol implementation. The group also previously reserved a set of physical machines for general login access (as opposed to reserved use by a specific research project). With Usher, a virtual cluster of convenience VMs now serves this purpose, and an alias with round-robin DNS provides a logical machine name for reference while distributing users among the VMs upon login. Even mundane tasks, such as experimenting with software installations or configurations, can benefit as well because the cost of creating a new machine is negligible. Rather than having to undo mistakes, a user can simply destroy a VM with an aborted configuration and start from scratch with a new one.

The SysNet group currently uses a simple policy module in Usher to determine the scheduling and

placement of VMs. This module relies upon monitoring data collected by the controller to make its decisions. It uses heuristics to place new VMs on lightly loaded physical machines, and to migrate VMs when a particular VM imposes sustained high load on a physical machine. Users are reasonably self-policing; they could always create large numbers of VMs to fully consume system resources, for example, but in practice do not. Eventually, as the utilization of physical machines increases to the point where VMs substantially interfere with each other, the group will interpret it as a signal that it is time to purchase additional hardware for the cluster.

This policy works well for the group, but of course is not necessarily suitable for all situations, such as the RRC-KI deployment described below in the RRC-KI section.

Support Services

Usher at UCSD uses plugins to automatically assigns IP addresses and VLANs to VMs, creates convenient domain name groupings for VMs in a virtual cluster, installs default user accounts, and provides structured VM-local, VC-global, and system-global file system access. These plugins interact with four support servers running as part of the site infrastructure.

SQL Server: The global state of the SysNet installation is kept in an SQL backing database. The database plugin mentioned in the Plugins subsection of the Implementation section provides access to the SQL database. Though most of the stored data is logging data stored for offline analysis of system performance and behavior, the SQL database does provide one required service: IP address management. The SysNet installation does not use DHCP to manage IP address ranges. The SysNet group manages several subnets, spanning multiple VLANs. Assigning ownership of arbitrary IP address ranges of these subnets to specified Usher users would be impossible using DHCP. As a result, an Usher plugin handles IP address management across these subnets.

LDAP Server: The SysNet LDAP plugin serves two purposes. First, it provides methods for managing and authenticating Usher users. Second, it provides the convenience of creating a branch in the LDAP database for each cluster an Usher user creates. This branch enables each VM the user creates to authenticate its users through the LDAP database.

This functionality provides a convenient authentication service to virtual cluster creators. First, it allows Usher users to use their Usher credentials as their VM login credentials since they are automatically added as a user in each cluster created. Since each cluster uses a different branch in the LDAP database, we use aliasing in LDAP to provide Usher users a single set of credentials. In addition, the plugin adds each Usher user to the "admin" group of each cluster the

user creates. VM filesystems can then be configured to grant special privileges to this group (e.g., sudo privileges). This approach is convenient when using a read-only NFS root filesystem where no default root password is set.

Second, and more importantly, this arrangement addresses the cluster authentication problem for Usher users in the SysNet group. Authentication for clusters is challenging enough for experienced administrators. Delegating this problem to users is not only time consuming for them, but could lead to insecure VMs.

Creating a separate branch for each cluster allows Usher users to create accounts and groups for their clusters without burdening the Usher administrator with this task. This capability is especially conducive to collaborative work, a common case in a research lab setting. An administrator could easily be overwhelmed with management requests in a setting where users are free to create their own clusters, yet are unable to fully manage them. This approach pushes many mundane administrative tasks out to the users who have the incentive to create accounts on their VMs.

Allowing Usher users to modify the LDAP database requires careful configuration of the LDAP server, however. An LDAP server configuration file that allows Usher users to only manage branches which they own is included with the Usher source code. In addition, the Usher plugin for the LDAP server includes scripts for installation on a user's VM filesystems to modify cluster LDAP entries (i.e., to add, modify, or delete users and groups).

DNS Server: By default, Usher names VMs using the following naming scheme:

```
<requested VM name>.<creator's username>.  
<Usher system domain name>
```

where the Usher system domain name is specified in a configuration file read by the controller at startup. The DNS plugin adds this name for both forward and reverse name resolution for each VM.

NAS Server: Live migration of virtual machines requires a filesystem accessible by the VM at both the source and destination VMM. Since migration is a requirement of the SysNet installation, SysNet VMs must have their root filesystems provided via network-attached storage. These filesystems are served read-only NFS.

Serving the root filesystem read-only has multiple benefits. First, it is straightforward to keep filesystems across all running VMs synchronized and updated using read-only NFS root filesystems. Furthermore, an experienced administrator can manage this filesystem to ensure that it is secure (e.g., default firewall rules, minimal services started by default, latest security patches, etc.).

Second, since all VMs mount this filesystem, it is important that it be as responsive as possible. Ensuring that the NFS server serving this filesystem is read-

only helps improve performance. Furthermore, an administrator can configure a read-only NFS server to cache the entire filesystem in main memory. As a result, reads go to disk only once.

One issue with using a read-only root filesystem is that some files and directories on the filesystem must be writable at system startup. We solve this problem using a ramdisk for any files and directories which must be writable. Early in the boot process, these files and directories are copied into the ramdisk, then mounted using the `--bind` flag to make them writable.

Since the SysNet installation serves its root filesystems read-only, another NFS server provides persistent writable storage. The filesystem plugin initializes the filesystems to be mounted prior to starting up a new VM. This plugin creates the following directories for each VM on the group's read-write NFS server:

- **/net/global:** This directory is where users install or store anything they would like to have globally accessible by all of their clusters. The contents of `/net/global` is the same for all VMs a user creates.
- **/net/cluster:** This directory is where users can store files they want accessible by the current cluster only. The contents of `/net/cluster` is the same for all VMs in the same cluster.
- **/net/local:** This directory is unique to the current VM only. The contents of `/net/local` is different for every VM a user creates. Users can use this directory to set up services and configuration files specific to particular VMs.

Finally, all SysNet users are given a home directory. Automount takes care of mounting these directories upon login. Alternatively, Usher users can choose an alternate URI (stored in LDAP) for their home directory.

In each of `/net/global`, `/net/cluster`, and `/net/local`, there exists a System V init style directory structure in the `etc` directory. Startup scripts in the directory for the appropriate runlevel are run from these three locations after the regular system startup scripts run. With this configuration, even though users cannot write to the root filesystem to change startup scripts, they can have services started for their VMs at VM boot.

RRC-KI

Usher has also been deployed at the Russian Research Center, Kurchatov Institute (RRC-KI). The RRC-KI deployment demonstrates the flexibility of Usher to integrate with different computing environments, and to employ different resource utilization policies. Whereas the UCSD SysNet Usher deployment targeted a general-purpose computing environment, the RRC-KI Usher deployment targets a batch job execution system that provides guaranteed resources to jobs.

RRC-KI contributes part of its compute infrastructure to the Large Hadron Collider (LHC) Grid

effort [1]. Scientists submit jobs to the system, which are scheduled via a batch job scheduler. Jobs are assigned to physical machines, and one machine only runs a single job at any time.

Measurements spanning over a year indicated that the overall utilization of machines in this system is fairly low [13]. While there were some long, compute intensive jobs, there was a large fraction of short, I/O driven jobs. Motivated by these measurements, the goal was to build a flexible job execution system that would improve the aggregate resource utilization of the cluster.

A straightforward approach is to multiplex several jobs on a single machine, and power down the unused machines. However, conventional process-based multiplexing on commodity operating systems is infeasible for a variety of reasons, some social and some technical: scientists want at least the appearance of absolute resource guarantees for their jobs; jobs often span multiple processes, which makes resource accounting and allocation challenging; and the number of physical machines needed depends on the workload and cannot be assigned *a priori*.

Virtual machines are a natural solution to this problem. Since each job gets its own isolated execution environment, resource accounting becomes easier for multi-process jobs. VMs also provide much stronger isolation guarantees than conventional processes. Each job can be given guaranteed resource reservations while still maintaining the abstraction of a physical machine. A trace-driven simulation showed that a VM-based infrastructure would enable significant savings [13].

One of the biggest challenges to this approach is management. For a VM-based infrastructure to scale, we need an automated system for deploying and managing virtual machines, a system that can schedule VMs in an intelligent manner, and migrate and place VMs to optimize utilization without sacrificing performance. A prototype system is currently being used at RRC-KI with Usher as the core management framework.

Central to this infrastructure is the *Policy Daemon* responsible for job scheduling and dynamically managing virtual machines (creation, migration, destruction) as a function of the current workload. The Policy Daemon uses the Usher Client API to monitor VM status and control VM resource utilization from a single control point using secure connections to the physical hosts. The current testbed comprises of a small number of nodes hosting production Grid jobs in the Usher-based environment with plans to expand the system to manage a few hundred nodes [15].

Adoption Considerations

Usher was designed to be a flexible, extensible framework for managing virtual cluster environments. However, our claims are supported only to the extent of what we have implemented and tested. At the time of this writing, we have used Usher with one VMM

implementation and the specific instances of plugins for UCSD and RRC-KI. For other sites to use Usher, if the existing plugins do not match their needs as implemented, then they will have to modify existing plugins or write their own. To this end, we do encourage Usher users to share any modified or new plugins they have implemented.

A final consideration is that of managing clusters of physical machines. Though the design of the framework does not preclude managing clusters of physical machines, to date, no plugins for managing physical clusters have been written.

Conclusions

Usher is an extensible, event-driven management system for clusters of virtual machines. The Usher core implements basic virtual machine and cluster management mechanisms, such as creating, destroying, and migrating VMs. Usher clients are applications that serve as user interfaces to the system, such as the interactive command-line shell *Ush*, as well as applications that use Usher as a foundation for creating and manipulating virtual machines for their own purposes. Usher supports customizable plugin modules for flexibly integrating Usher into other administrative services at a site, and for installing policies for the use, placement, and scheduling of virtual machines according to the site-specific requirements. Usher has been in production use both at UCSD and at the Russian Research Center in Kurchatov, Russia, and initial feedback from both users and administrators indicates that Usher is successfully achieving its goals.

Usher is free software distributed under the new BSD license. Source code, documentation, and tutorials are available at <http://usher.ucsd.edu>. Source code, configuration files, and initialization scripts for the UCSD plugins are also available for download at the site above.

Acknowledgments

The authors would like to thank Roman Kurakin for his insight, patches, and administration of Usher at RRC-KI. We also want to thank those people using Usher for their research at UCSD. Their feedback has been invaluable to the success of Usher in a research and academic environment. Finally, we would like to thank Alva Couch and our anonymous reviewers for their time and insightful comments regarding this paper. Support for this work was provided in part by NSF under CSR-PDOS Grant No. CNS-0615392 and the UCSD Center for Networked Systems.

Author Biographies

Marvin McNett is a Ph.D. student in the Systems and Networking group at the University of California, San Diego. His current research focus is virtual machine scheduling and management for efficient resource

utilization. He is the original developer and current maintainer of the Usher project. Marvin expects to finish his Ph.D. in December, 2007.

Diwaker Gupta is a Ph.D. student in the Systems and Networking group at the University of California, San Diego. His current research interests include resource management and performance isolation mechanisms in virtual machines.

Amin Vahdat is a Professor in the Department of Computer Science and Engineering and the Director of the Center for Networked Systems at the University of California San Diego. He received his Ph.D. in Computer Science from UC Berkeley in 1998. Before joining UCSD in January 2004, he was on the faculty at Duke University from 1999-2003.

Geoffrey M. Voelker is an Associate Professor at the University of California at San Diego. His research interests include operating systems, distributed systems, networking, and wireless networks. He received a B.S. degree in Electrical Engineering and Computer Science from the University of California at Berkeley in 1992, and the M.S. and Ph.D. degrees in Computer Science and Engineering from the University of Washington in 1995 and 2000, respectively.

Bibliography

- [1] *LCG project*, <http://lcg.web.cern.ch/LCG/>.
- [2] *Python*, <http://www.python.org/>.
- [3] *Twisted*, <http://twistedmatrix.com/>.
- [4] *VirtualCenter*, <http://www.vmware.com/products/vi/vc/>.
- [5] *XenEnterprise*, http://www.xensource.com/products/xen_enterprise/.
- [6] Albrecht, Jeannie, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, "Remote Control: Distributed Application Configuration, Management, and Visualization with Plush," *Proceedings of the Twenty-first USENIX Large Installation System Administration Conference (LISA)*, November, 2007.
- [7] Albrecht, Jeannie, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, "PlanetLab Application Management Using Plush," *ACM Operating Systems Review (SIGOPS-OSR)*, Vol. 40, Num. 1, January, 2006.
- [8] Anderson, Thomas E., David E. Culler, David A. Patterson, and the NOW Team, "A Case for Networks of Workstations: NOW," *IEEE Micro*, February, 1995.
- [9] Appleby, K., S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Océano - SLA-based Management of a Computing Utility," *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May, 2001.
- [10] Begnum, Kyrre, "Managing Large Networks of Virtual Machines," *Proceedings of the 20th Large*

- Installation System Administration Conference*, pp. 205-214, 2006.
- [11] Bruno, G., M. J. Katz, F. D. Sacerdoti, and P. M. Papadopoulos, "Rolls: Modifying a Standard System Installer to Support User-customizable Cluster Frontend Appliances," *IEEE International Conference on Cluster Computing*, 2004.
 - [12] Chase, Jeffrey S., David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle, "Dynamic Virtual Clusters in a Grid Site Manager," *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.
 - [13] Cherkasova, Ludmila, Diwaker Gupta, Roman Kurakin, Vladimir Dobretsov, and Amin Vahdat, "Optimising Grid Site Manager Performance With Virtual Machines," *Proceedings of the 3rd USENIX Workshop on Real Large Distributed Systems (WORLD5)*, 2006.
 - [14] Grit, Laura, David Irwin, Aydan Yumerefendi, and Jeff Chase, "Harnessing Virtual Machine Resource Control for Job Management," *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November, 2006.
 - [15] Kurakin, Roman, Personal communication, Email dated 5/10/2007.
 - [16] Merkey, Phil, *Beowulf History*, <http://www.beowulf.org/overview/history.html>.
 - [17] Ruth, P., Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen, "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," *IEEE International Conference on Autonomic Computing*, June, 2006.
 - [18] Sacerdoti, F. D., S. Chandra, and K. Bhatia, "Grid Systems Deployment and Management Using Rocks," *IEEE International Conference on Cluster Computing*, 2004.
 - [19] Vogels, Werner and Dan Dumitriu, "An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing," *Proceedings of the IEEE International Conference on Cluster Computing*, 2000.
 - [20] Waldspurger, Carl A., "Memory Resource Management in VMware ESX Server," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, December, 2002.
 - [21] White, Brian, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 255-270, USENIX Association, Boston, MA, December, 2002.
 - [22] Wood, Timothy, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," *Proceedings the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, April, 2007.

Remote Control: Distributed Application Configuration, Management, and Visualization with Plush

Jeannie Albrecht – Williams College

*Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren,
and Amin Vahdat* – University of California, San Diego

ABSTRACT

Support for distributed application management in large-scale networked environments remains in its early stages. Although a number of solutions exist for subtasks of application deployment, monitoring, maintenance, and visualization in distributed environments, few tools provide a unified framework for application management. Many of the existing tools address the management needs of a single type of application or service that runs in a specific environment, and these tools are not adaptable enough to be used for other applications or platforms. In this paper, we present the design and implementation of Plush, a fully configurable application management infrastructure designed to meet the general requirements of several different classes of distributed applications and execution environments. Plush allows developers to specifically define the flow of control needed by their computations using application building blocks. Through an extensible resource management interface, Plush supports execution in a variety of environments, including both live deployment platforms and emulated clusters. To gain an understanding of how Plush manages different classes of distributed applications, we take a closer look at specific applications and evaluate how Plush provides support for each.

Introduction

Managing distributed applications involves deploying, configuring, executing, and debugging software running on multiple computers simultaneously. Particularly for applications running on resources that are spread across the wide-area, distributed application management is a time-consuming and error-prone process. After the initial deployment of the software, the applications need mechanisms for detecting and recovering from the inevitable failures and problems endemic to distributed environments. To achieve availability and reliability, applications must be carefully monitored and controlled to ensure continued operation and sustained performance. Operators in charge of deploying and managing these applications face a daunting list of challenges: discovering and acquiring appropriate resources for hosting the application, distributing the necessary software, and appropriately configuring the resources (and re-configuring them if operating conditions change). It is not surprising, then, that a number of tools have been developed to address various aspects of the process in distributed environments, but no solution yet flexibly automates the application deployment and management process across all environments.

Presently, most researchers who want to evaluate their applications in wide-area distributed environments take one of three management approaches. On PlanetLab [6, 26], service operators address deployment and monitoring in an *ad hoc*, application-specific

fashion using customized scripts. Grid researchers, on the other hand, leverage one or more toolkits (such as the Globus Toolkit [12]) for application development and deployment. These toolkits often require tight integration with not only the infrastructure, but the application itself. Hence, applications must be custom tailored for a given toolkit, and can not easily be run in other environments. Similarly, system administrators who are responsible for configuring resources in machine-room settings often use remote execution tools such as cfengine [9] for managing and configuring networks of machines. As in the other two approaches, however, the configuration files are tailored to a specific environment and a particular set of resources, and thus are not easily extended to other platforms.

Motivated by the limitations of existing approaches, we believe that a unified set of abstractions for achieving availability, scalability, and fault tolerance can be applied to a broad range of distributed applications, shielding developers from some of the complexities of large-scale networked environments. The primary goal of our research is to understand these abstractions and define interfaces for specifying and managing distributed computations run in *any* execution environment. We are not trying to build another toolkit for managing distributed applications. Rather, we hope to define the way users think about their applications, regardless of their target platform. We took inspiration from classical

operating systems like UNIX [28] which defined the standard abstractions for managing applications: files, processes, pipes, etc. For most users, communication with these abstractions is simplified through the use of a shell or command-line interpreter. Of course, distributed computations are both more difficult to specify, because of heterogeneous hardware and software bases, and more difficult to manage, because of failure conditions and variable host and network attributes. Further, many distributed testbeds do not provide global file system abstractions, which complicates data management.

To this end, we present Plush [27], a generic application management infrastructure that provides a unified set of abstractions for specifying, deploying, and monitoring distributed applications. Although Plush was initially designed to support applications running on PlanetLab [2], Plush now provides extensions that allow users to manage distributed applications in a variety of computing environments. Plush users describe distributed computations using an extensible application specification language. In contrast to other application management systems, however, the language allows users to customize various aspects of the deployment life cycle to fit the needs of an application and its target infrastructure. Users can, for example, specify a particular resource discovery service to use during application deployment. Plush also provides extensive failure management support to automatically adapt to failures in the application and the underlying computational infrastructure. Users interact with Plush through a simple command-line interface or a graphical user interface (GUI). Additionally, Plush exports an XML-RPC interface that allows users to programmatically integrate their applications with Plush if desired.

Plush provides abstractions for managing resource discovery and acquisition, software distribution, and process execution in a variety of distributed environments. Applications are specified using combinations of Plush application "building blocks" that define a custom control flow. Once an application is running, Plush monitors it for failures or application-level errors for the duration of its execution. Upon detecting a problem, Plush performs a number of user-configurable recovery actions, such as restarting the application, automatically reconfiguring it, or even searching for alternate resources. For applications requiring wide-area synchronization, Plush provides several efficient synchronization primitives in the form of partial barriers, which help applications achieve better performance and robustness in failure-prone environments [1].

The remainder of this paper discusses the architecture of Plush. We motivate the design in the next section by enumerating a set of general requirements for managing distributed applications. Subsequently, we present details about the design and implementation of Plush and then provide specific application case studies and uses of Plush. Related work is shown in the next section which is followed by the conclusion.

Application Management Requirements

To better understand the requirements of a distributed application management framework, we first consider how we might run a specific application in a widely-used distributed environment. In particular, we investigate the process of running SWORD [23], a publicly-available resource discovery service, on PlanetLab. SWORD uses a distributed hash table (DHT) for storing data, and aims to run on as many hosts as possible, as long as the hosts provide some minimum level of availability (since SWORD provides a service to other PlanetLab users). Before starting SWORD, we have to find and gain access to PlanetLab machines capable of hosting the service. Since SWORD is most concerned with reliability, it does not necessarily need powerful machines, but it must avoid nodes that frequently perform poorly over a relatively long time. We locate reliable machines using a tool like CoMon [25], which monitors resource usage on PlanetLab, and then we install the SWORD software on those machines.

This software installation involves downloading the SWORD software package on each host individually, unpacking the software, and installing any software dependencies, including a Java Runtime Environment. After the software has been installed on all of the selected machines, we start the SWORD execution. Recall that reliability is important to SWORD, so if an error or failure occurs at any point, we need to quickly detect it (perhaps using custom scripts and cron jobs) and restore the service to maintain high availability.

Running SWORD on PlanetLab is an example of a specific distributed application deployment. The low-level details of managing distributed applications *in general* largely depend on the characteristics of the target application and environment. For example, long-running services such as SWORD prefer reliable machines and attempt to dynamically recover from failures to ensure high availability. On the other hand, short-lived scientific parallel applications (e.g., EMAN [18]) prefer powerful machines with high bandwidth/low latency network connections. Long term reliability is not a huge concern for these applications, since they have short execution times. At a high level, however, if we ignore the complexities associated with resource management, the requirements for managing distributed applications are largely similar for all applications and environments. Rather than reinvent the same infrastructure for each class separately, our goal is to identify common abstractions that support the execution of many types of distributed applications, and to build an application-management infrastructure that supports the general requirements of all applications. In this section, we identify these general requirements for distributed application management.

Specification. A generic application controller must allow application operators to customize the

control flow for each application. This *specification* is an abstraction that describes distributed computations. A specification identifies all aspects of the execution and environment needed to successfully deploy, manage, and maintain an application, including the software required to run the application, the processes that will run on each machine, the resources required to achieve the desired performance, and any environment-specific execution parameters. User credentials for resources must also be included in the application specification in order to obtain access to resources. To manage complex multi-phased computations, such as scientific parallel applications, the specification must support application synchronization requirements. Similarly, distributing computations among pools of machines requires a way to specify a workflow – a collection of tasks that must be completed in a given order – within an application specification.

The complexity of distributed applications varies greatly from simple, single-process applications to elaborate, parallel applications. Thus the challenge is to define a specification language abstraction that provides enough expressibility for complex distributed applications, but is not too complicated for single-process computations. In short, the language must be simple enough for novice application developers to understand, yet expose enough advanced functionality to run complex scenarios.

Resource Discovery and Acquisition. Another key abstraction in distributed applications are *resources*. Put simply, resources are computing devices that are connected to a network and are capable of hosting an application. Because resources in distributed environments are often heterogeneous, application developers naturally want to find the resource set that best satisfies the demands of their application. Even if hardware is largely homogeneous, dynamic resource characteristics such as available bandwidth or CPU load can vary over time. The goal of resource discovery is to find the best *current* set of resources for the distributed application as described in the specification. In environments that support dynamic virtual machine instantiation [5, 30], these resources may not exist in advance. Thus, resource discovery in this case involves finding the appropriate physical machines to host the virtual machine configurations.

Resource discovery systems often interact directly with resource acquisition systems. Resource acquisition involves obtaining a lease or permission to use the desired resources. Depending on the execution environment, acquisition can take a number of forms. For example, to support advanced resource reservations as in a batch pool, resource acquisition is responsible for submitting a resource request and subsequently obtaining a lease from the scheduler. In virtual machine environments, resource acquisition may involve instantiating virtual machines, verifying their successful creation, and gathering the appropriate information (e.g., IP

address, authentication keys) required for access. The challenge facing an application management framework is to provide a generic resource-management interface. Ultimately, the complexities associated with creating and gaining access to physical or virtual resources should be hidden from the application developer.

Deployment. Upon obtaining an appropriate set of resources, the application-deployment abstraction defines the steps required to prepare the resources with the correct software and data files, and run any necessary executables to start the application. This involves copying, unpacking, and installing the software on the target hosts. The application controller must support a variety of different file-transfer mechanisms for each environment, and should react to failures that occur during the transfer of software or in starting executables.

One important aspect of application deployment is configuring the requested number of resources with compatible versions of the software. Ensuring that a minimum number of resources are available and correctly configured for a computation may involve requesting new resources from the resource discovery and acquisition systems to compensate for failures that occur at startup. Further, many applications require some form of synchronization across hosts to guarantee that various phases of computation start at approximately the same time. Thus, the application controller must provide mechanisms for loose synchronization.

Maintenance. Perhaps the most difficult requirement for managing distributed applications is monitoring and maintaining an application after execution begins. Thus, another abstraction that the application controller must define is support for customizable application maintenance. One key aspect of maintenance is application and resource monitoring, which involves probing hosts for failure due to network outages or hardware malfunctions, and querying applications for indications of failure (often requiring hooks into application-specific code for observing the progress of an execution). Such monitoring allows for more specific error reporting and simplifies the debugging process.

In some cases, system failures may result in a situation where application requirements can no longer be met. For example, if an application is initially configured to be deployed on 50 resources, but only 48 can be contacted at a certain point in time, the application controller should adapt the application, if possible, and continue executing with only 48 machines. Similarly, different applications have different policies and requirements with respect to failure recovery. Some applications may be able to simply restart a failed process on a single host, while others may require the entire execution to abort in the case of failure. Thus, in addition to the other features previously described, the application controller should support a variety of options for failure recovery.

Plush: Design and Implementation

We now describe Plush, an extensible distributed application controller, designed to address the requirements of large-scale distributed application management discussed in the second section.

To directly monitor and control distributed applications, Plush itself must be distributed. Plush uses a client-server architecture, with clients running on each resource (e.g., machine) involved in the application. The Plush server, called the *controller*, interprets input from the user (i.e., the person running the application) and sends messages on behalf of the user over an overlay network (typically a tree) to Plush *clients*. The

controller, typically run from the user's workstation, directs the flow of control throughout the life of the distributed application. The clients run alongside each application component across the network and perform actions based upon instructions received from the controller.

Figure 1a shows an overview of the Plush controller architecture. (The client architecture is symmetric to the controller with only minor differences in functionality.) The architecture consists of three main sub-systems: the application specification, core functional units, and user interface. Plush parses the application specification provided by the user and stores

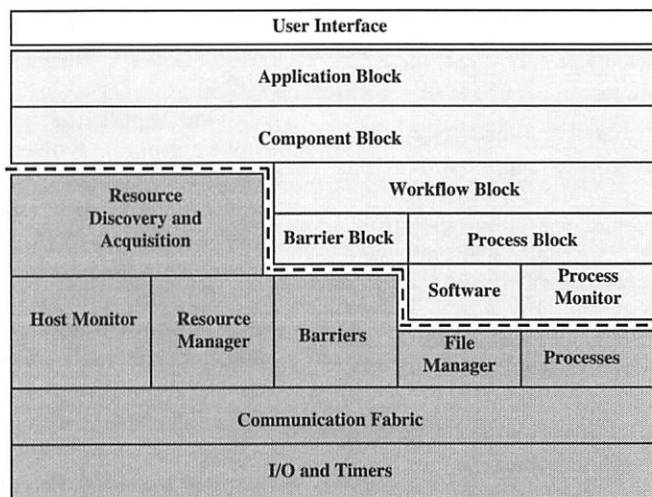


Figure 1a: The architecture of Plush. The *user interface* is shown above the rest of the architecture and contains methods for interacting with all boxes in the lower sub-systems of Plush. Boxes below the user interface and above the dotted line indicate objects defined within the *application specification* abstraction. Boxes below the line represent the *core functional units* of Plush.

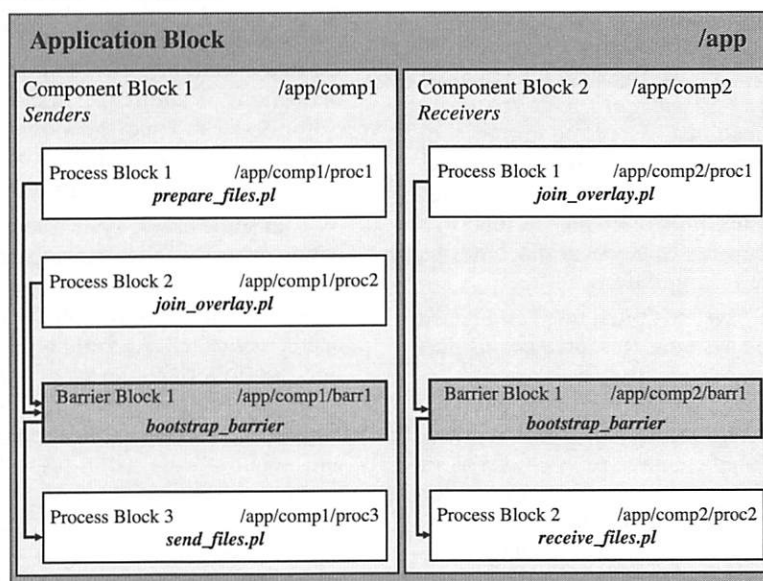


Figure 1b: Example file-distribution application comprised of application, component, process, and barrier blocks in Plush. Arrows indicate control-flow dependencies (i.e., $X \rightarrow Y$ implies that X must complete before Y starts).

internal data structures and objects specific to the application being run. The core functional units then manipulate and act on the objects defined by the application specification to run the application. The functional units also store authentication information, monitor physical machines, handle event and timer actions, and maintain the communication infrastructure that enables the controller to query the status of the distributed application on the clients. The user interface provides the functionality needed to interact with the other parts of the architecture, allowing the user to maintain and manipulate the application during execution. In this section, we describe the design and implementation details of each of the Plush sub-systems.¹

Application Specification

Developing a complete, yet accessible, application specification language was one of the principal challenges in this work. Our approach, which has evolved over the past three years, consists of combinations of five different abstractions:

1. **Process blocks** – Describe the processes executed on each machine involved in an application. The process abstraction includes runtime parameters, path variables, runtime environment details, file and process I/O information, and the specific commands needed to start a process on a remote machine.
2. **Barrier blocks** – Describe the barriers that are used to synchronize the various phases of execution within a distributed application.
3. **Workflow blocks** – Describe the flow of data in a distributed computation, including how the data should be processed. Workflow block may contain process and barrier blocks. For example, a workflow block might describe a set of input files over which a process or barrier block will iterate during execution.
4. **Component blocks** – Describe the groups of resources required to run the application. This includes expectations specific to a set of metrics for the target resources. In the case of compute nodes in a cluster, for example, these metrics might include maximum load requirements and minimum free memory requirements. Components also define required software configurations, installation instructions, and any authentication information needed to access the resources. Component blocks may contain workflow blocks, process blocks, and barrier blocks.
5. **Application blocks** – Describe high-level information about a distributed application. This includes one or many component blocks, as well as attributes to help automate failure recovery.

To better illustrate the use of these blocks in Plush, consider building the specification for a simple

file-distribution application as shown in Figure 1b. This application consists of two groups of machines. One group, the senders, stores the files, and the second group, the receivers, attempt to retrieve the files from the senders. The goal of the application is to experiment with the use of an overlay network to send files from the senders to the receivers using some new file-distribution protocol. In this example, there are two phases of execution. In the first phase, all senders and receivers join the overlay before any transfers begin, and the senders must prepare the files for transfer before the receivers start receiving files. In the second phase, the receivers begin receiving the files. No new senders or receivers are allowed to join the network during the second phase.

The first step in building the corresponding Plush application specification for our new file-distribution protocol is to define an application block. The application block defines general characteristics about the application including liveness properties and failure detection and recovery options, which determine default failure recovery behavior. For this example, we choose the behavior “restart-on-failure,” which attempts to restart the failed application instance on a single host, since it is not necessary to abort the entire application across all hosts if only a single failure occurs.

The application block also contains one or many component blocks that describe the groups of resources (i.e., machines) required to run the application. Our application consists of a set of senders and a set of receivers, and two separate component blocks describe the two groups of machines. The sender component block defines the location and installation instructions for the sender software, and includes authentication information to access the resources. Similarly, the receiver component block defines the receiver software package. In our example, it may be desirable to require that all machines in the sender group have a processor speed of at least 1 GHz, and each sender should have sufficient bandwidth for sending files to multiple receivers at once. These types of machine-specific requirements are included in the component blocks. Within each component block, a combination of workflow, process, and barrier blocks describe the distributed computation.²

Plush process blocks describe the specific commands required to execute the application. Most process blocks depend on the successful installation of **software** packages defined in the component blocks. Users specify the commands required to start a given process, and actions to take upon process exit. The exit policies create a Plush **process monitor** that oversees the execution of a specific process. Our example has several process blocks. In the sender component, process blocks define processes for preparing the files, joining the overlay, and

¹Note that the components within the sub-systems are highlighted using boldface throughout the text in the remainder of this section.

²Although our example does not use workflow blocks, they are used in applications where data files must be distributed and iteratively processed.

sending the files. Similarly, the receiver component contains process blocks for joining the overlay and receiving the files.

Some applications operate in phases, producing output files in early stages that are used as input files in later stages. To ensure all hosts start each phase of computation only after the previous phase completes, barrier blocks define loose synchronization semantics between process and workflow blocks. In our example, a barrier ensures that all receivers and senders join the overlay in phase one before beginning the file transfer in phase two. Note that although each barrier block is uniquely defined within a component block, it is possible for the same barrier to be referenced in multiple component blocks. We use barrier blocks in our example within each component block that refer to the same barrier, which means that the application will wait for all receivers and senders to reach the barrier before allowing either component to start sending or receiving files.

In Figure 1b, the outer application block contains our two component blocks that run in parallel (since there are no arrows indicating control-flow dependencies between them). Within the component blocks, the different phases are separated by the bootstrap barrier that is defined by barrier block 1. Component block 1, which describes the senders, contains process blocks 1 and 2 that define perl scripts that run in parallel during phase one, synchronize on the barrier in barrier block 1, and then proceed to process block 3 in phase two which sends the files. Component block 2, which describes the receivers, runs process block 1 in phase one, synchronizes on the barrier in barrier block 1, and then proceeds to process block 2 in phase two which runs the process that receives the files. In our implementation, the blocks are represented by XML that is parsed by the Plush controller when the application is run. We show an example of the XML later.

We designed the Plush application specification to support a variety of execution patterns. With the blocks described above, Plush supports the arbitrary combination of processes, barriers, and workflows, provided that the flow of control between them forms a directed acyclic graph. Using predecessor tags in Plush, users specify the flow of control and define whether processes run in parallel or sequentially. Arrows between blocks in Figure 1b, for example, indicate the predecessor dependencies. (Process blocks 1 and 2 in component block 1 will run in parallel before blocking at the bootstrap barrier, and then the execution will continue on to process block 3 after the bootstrap barrier releases.) Internally, Plush stores the blocks in a hierarchical data structure, and references specific blocks in a manner similar to referencing absolute paths in a UNIX file system. Figure 1b shows the unique path names for each block from our file-distribution example. This naming abstraction also simplifies coordination among remote hosts. Each

Plush client maintains an identical local copy of the application specification. Thus, for communication regarding control flow changes, the controller sends the clients messages indicating which "block" is currently being executed, and the clients update their local state information accordingly.

Core Functional Units

After parsing the block abstractions defined by the user within the application specification, Plush instantiates a set of core functional units to perform the operations required to configure and deploy the distributed application. Figure 1a shows these units as shaded boxes below the dotted line. The functional units manipulate the objects defined in the application specification to manage distributed applications. In this section, we describe the role of each of these units.

Starting at the highest level, the Plush **resource discovery and acquisition** unit uses the resource requirements in the component blocks to locate and create (if necessary) resources on behalf of the user. The resource discovery and acquisition unit is responsible for obtaining a valid set, called a *matching*, of resources that meet the application's demands. To determine this matching, Plush may either call an existing external service to construct a resource pool, such as SWORD or CoMon for PlanetLab, or use a statically defined resource pool based on information provided by the user. The Plush *resource matcher* then uses the resources in the resource pool to create a matching for the application. All hosts involved in an application run a Plush **host monitor** that periodically publishes information about the host. The resource discovery and acquisition unit may use this information to help find the best matching. Upon acquiring a resource, a Plush **resource manager** stores the lease, token, or any necessary user credential needed for accessing that resource to allow Plush to perform actions on behalf of the user in the future.

The remaining functional units in Figure 1a are responsible for application deployment and maintenance. These units connect to resources, install required software, start the execution, and monitor the execution for failures. One important functional unit used for these operations is the Plush **barrier manager**, which provides advanced synchronization services for Plush and the application itself. In our experience, traditional barriers [17] are not well suited for volatile, wide-area network conditions; the semantics are simply too strict. Instead, Plush uses partial barriers, which are designed to perform better in volatile environments [1]. Partial barriers ensure that the execution makes forward progress in the face of failures, and improve performance in failure-prone environments using relaxed synchronization semantics.

The Plush **file manager** handles all files required by a distributed application. This unit contains information regarding software packages, file transfer methods,

installation instructions, and workflow data files. The file manager is responsible for preparing the physical resources for execution using the information provided by the application specification. It monitors the status of file transfers and installations, and if it detects an error or failure, the controller is notified and the resource discovery and acquisition unit may be required to find a new host to replace the failed one.

Once the resources are prepared with the necessary software, the application deployment phase completes by starting the execution. This is accomplished by starting a number of processes on remote hosts. Plush **processes** are defined within process blocks in the application specification. A Plush process is an abstraction for standard UNIX processes that run on multiple hosts. Processes require information about the runtime environment needed for an execution including the working directory, path, environment variables, file I/O, and the command line arguments.

The two lowest layers of the Plush architecture consist of a **communication fabric** and the **I/O and timer** subsystems. The communication fabric handles passing and receiving messages among Plush overlay participants. Participants communicate over TCP connections. The default topology for a Plush overlay is a star, although we also provide support for tree topologies for increased scalability (discussed later in detail). In the case of a star topology, all clients connect directly to the controller, which allows for quick failure detection and recovery. The controller sends messages to the clients instructing them to perform certain actions. When the clients complete their tasks, they report back to the controller for further direction. The communication fabric at the controller knows what hosts are involved in a particular application instance, so that the appropriate messages reach all necessary hosts.

At the bottom of all of the other units is the Plush I/O and timer abstraction. As messages are received in the communication fabric, message handlers fire events. These events are sent to the I/O and timer layer and enter a queue. The event loop pulls events off the queue, and calls the appropriate event handler. Timers are a special type of event in Plush that fire at a predefined instant.

Fault Tolerance and Scalability

Two of the biggest challenges that we encountered during the design of Plush was being robust to failures and scaling to hundreds of machines spread across the wide-area. In this section we explore fault tolerance and scalability in Plush.

Fault Tolerance

Plush must be robust to the variety of failures that occur during application execution. When designing Plush, we aimed to provide the functionality needed to detect and recover from most failures without ever needing to involve the user running the application. Rather than enumerate all possible failures that

may occur, we will discuss how Plush handles three common failure classes – process, host, and controller failures.

Process failures. When a remote host starts a process defined in a process block, Plush attaches a process monitor to the process. The role of the process monitor is to catch any signals raised by the process, and to react appropriately. When a process exits either due to successful completion or error, the process monitor sends a message to the controller indicating that the process has exited, and includes its exit status. Plush defines a default set of behaviors that occur in response to a variety of exit codes (although these can be overridden within an application specification). The default behaviors include ignoring the failure, restarting only the failed process, restarting the application, or aborting the entire application.

In addition to process failures, Plush also allows users to monitor the status of a process that is still running through a specific type of process monitor called a **liveness monitor**, whose goal is to detect misbehaving and unresponsive processes that get stuck in loops and never exit. This is especially useful in the case of long-running services that are not closely monitored by the user. To use the liveness monitor, the user specifies a script and a time interval in the process block of the application specification. The liveness monitor wakes up once per time interval and runs the script to test for the liveness of the application, returning either success or failure. If the test fails, the Plush client kills the process, causing the process monitor to be alerted and inform the controller.

Remote host failures. Detecting and reacting to process failures is straightforward since the controller is able to communicate information to the client regarding the appropriate recovery action. When a host fails, however, recovering is more difficult. A host may fail for a number of reasons, including network outages, hardware problems, and power loss. Under all of these conditions, the goal of Plush is to quickly detect the problem and reconfigure the application with a new set of resources to continue execution. The Plush controller maintains a list of the last time successful communication occurred with each connected client. If the controller does not hear from a client within a specified time interval, the controller sends a ping to the client. If the controller does not receive a response from the client, we assume host failure. Reliable failure detection is an active area of research; while the simple technique we employ has been sufficient thus far, we intend to leverage advances in this space where appropriate.

There are three possible actions in response to a host failure: restart, rematch, and abort. By default, the controller tries all three actions in order. The first and easiest way to recover from a host failure is to simply reconnect and restart the application on the failed host.

This technique works if the host experiences a temporary power or network outage, and is only unreachable for a short period of time. If the controller is unable to reconnect to the host, the next option is to rematch in an attempt to replace the failed host with a different host. In this case, Plush reruns the resource matcher to find a new machine. Depending on the application, the entire execution may need to be restarted across all hosts after the new host joins the control overlay, or the execution may only need to be started on the new host. If the controller is unable to find a new host to replace the failed host, Plush aborts the entire application.

In some applications, it is desirable to mark a host as failed when it becomes overloaded or experiences poor network connectivity. The Plush host monitor that runs on each machine is responsible for periodically informing the controller about each machine's status. If the controller determines that the performance is less than the application tolerates, it marks the host as failed and attempts to rematch. This functionality is a preference specified at startup. Although Plush currently monitors host-level metrics including load and free memory, the technique is easily extended to encompass sophisticated application-level expectations of host viability.

Controller failures. Because the controller is responsible for managing the flow of control across all connected clients, recovering from a failure at the controller is difficult. One solution is to use a simple primary-backup scheme, where multiple controllers increase reliability. All messages sent from the clients and primary controller are sent to the backup controllers as well. If a pre-determined amount of time passes and the backup controllers do not receive any messages from the primary, the primary is assumed to have failed. The first backup becomes the primary, and execution continues.

This strategy has several drawbacks. First, it causes extra messages to be sent over the network, which limits the scalability of Plush. Second, this approach does not perform well when a network partition occurs. During a network partition, multiple controllers may become the primary controller for subsets of the clients initially involved in the application. Once the network partition is resolved, it is difficult to reestablish consistency among all hosts. While we have implemented this architecture, we are currently exploring other possibilities.

Scalability

In addition to fault tolerance, an application controller designed for large-scale environments must scale to hundreds or even thousands of participants. Unfortunately there is a tradeoff between performance and scalability. The solutions that perform the best at moderate scale typically do not scale as well as solutions with lower performance. To balance scalability and performance, Plush provides users with two topological alternatives.

By default, all Plush clients connect directly to the controller forming a star topology. This architecture scales to approximately 300 remote hosts, limited by the number of file descriptors allowed per process on the controller machine in addition to the bandwidth, CPU, and latency required to communicate with all connected clients. The star topology is easy to maintain, since all clients connect directly to the controller. In the event of a host failure, only the failed host is affected. Further, the time required for the controller to exchange messages with clients is short due to the direct connections.

At larger scales, network and file descriptor limitations at the controller become a bottleneck. To address this, Plush also supports tree topologies. In an effort to reduce the number of hops between the clients and the controller, we construct "bushy" trees, where the depth of the tree is small and each node in the tree has many children. The controller is the root of the tree. The children of the root are chosen to be well-connected and historically reliable hosts whenever possible. Each child of the root acts as a "proxy controller" for the hosts connected to it. These proxy controllers send invitations and receive joins from other hosts, reducing the total number of messages sent back to the root controller. Important messages, such as failure notifications, are still sent back to the root controller. Using the tree topology, we have been able to use Plush to manage an application running on 1000 ModelNet [29] emulated hosts, as well as an application running on 500 PlanetLab clients. We believe that Plush has the ability to scale by another order of magnitude with the current design.

While the tree topology has many benefits over the star topology, it also introduces several new problems with respect to host failures and tree maintenance. In the star topology, a host failure is simple to recover from since it only involves one host. In the tree topology, however, if a non-leaf host fails, all children of the failed host must find a new parent. Depending on the number of hosts affected, a reconfiguration involving several hosts often has a significant impact on performance. Our current implementation tries to minimize the probability of this type of failure by making intelligent decisions during tree construction. For example, in the case of ModelNet, many virtual hosts (and Plush clients) reside on the same physical machine. When constructing the tree in Plush, only one client per physical machine connects directly to the controller and becomes the proxy controller. The remaining clients running on the same physical machine become children of the proxy controller. In the wide area, similar decisions are made by placing hosts that are geographically close together under the same parent. This decreases the number of hops and latency between leaf nodes and their parent, minimizing the chance of network failures.

Running An Application Using Plush

In this section, we will discuss how the architectural components of Plush interact to run a distributed application. When starting Plush, the user's workstation becomes the controller. The user submits an application specification to the Plush controller. The controller parses the specification, and internally creates the objects shown above the dotted line in Figure 1a.

After parsing the application specification, the controller runs the resource discovery and acquisition unit to find a suitable set of resources that meet the requirements specified in the component blocks. Upon locating the necessary resources, the resource manager stores the required access and authentication information. The controller then attempts to connect to each

remote host. If the Plush client is not already running, the controller initiates a bootstrapping procedure to copy the Plush client binary to the remote host, and then uses SSH to connect to the remote host and start the client process. Once the client process is running, the controller establishes a TCP connection to the remote host, and transmits an INVITE message to the host to join the Plush overlay (which is either a star or tree as discussed previously).

If a Plush client agrees to run the application, the client sends a JOIN message back to the controller accepting the invitation. Next, the controller sends a PREPARE message to the new client, which contains a copy of the application specification (XML representation). The client parses the application specification,

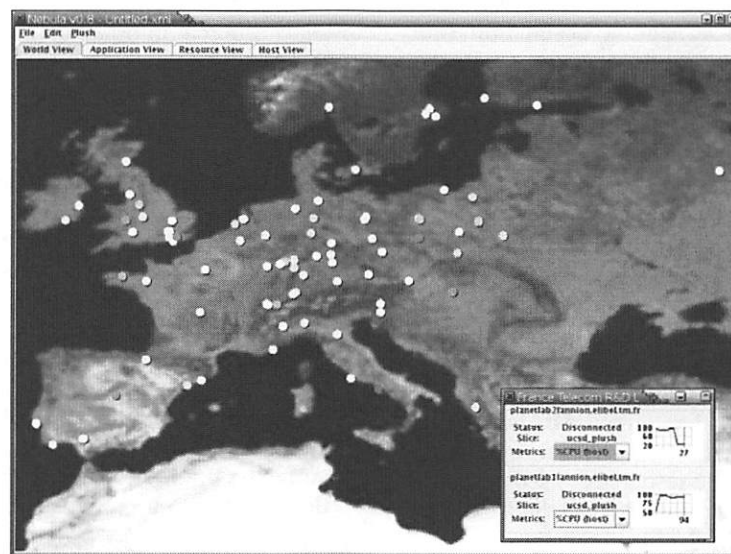


Figure 2a: Nebula World View tab showing an application running on PlanetLab. Different colored dots indicate PlanetLab sites in various stages of execution.

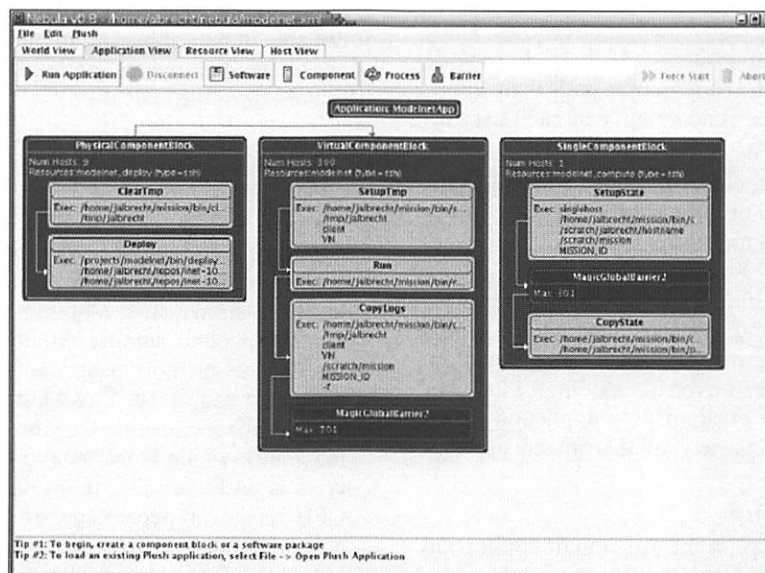


Figure 2b: Nebula Application View tab displaying Plush application specification.

starts a local host monitor, sends a PREPARED message back to the controller, and waits for further instruction. Once enough hosts join the overlay and agree to run the application, the controller initiates the beginning of the application deployment stage by sending a GO message to all connected clients. The file managers then begin installing the requested software and preparing the hosts for execution.

In most applications, the controller instructs the hosts to begin execution after all hosts have completed the software installation. (Synchronizing the beginning of the execution is not required if the application does not need all hosts to start simultaneously.) Since each client has now created an exact copy of the controller's application specification, the controller and clients exchange messages about the application's progress using the block naming abstraction (i.e., /app/comp1/procl) to identify the status of the execution. For barriers, a barrier manager running on the controller determines when it is appropriate for hosts to be released from the barriers.

Upon detecting a failure, clients notify the controller, and the controller attempts to recover from it according to the actions enumerated in the user's application specification. Since many failures are application-specific, Plush exports optional callbacks to the application itself to determine the appropriate reaction for some failure conditions. When the application completes (or upon a user command), Plush stops all associated processes, transfers output data back to the controller's local disk if desired, performs user-specified cleanup actions on the resources, disconnects the resources from the overlay by closing the TCP connections, and stops the Plush client processes.

User Interface

Plush aims to support a variety of applications being run by users with a wide range of expertise in building and managing distributed applications. Thus, Plush provides three interfaces which each provide users with techniques for interacting with their applications. We describe the functionality of each user interface in this section.

Figure 1a shows the user interface above all other parts of Plush. In reality, the user interacts with every box shown in the figure through the user interface. For example, the user forces the resource discovery and acquisition unit to find a new set of resources using a terminal command. We designed Plush in this way to provide maximum control over the application. At any point, the user can override a default Plush behavior. The effect is a customizable application controller that supports a variety of distributed applications.

Graphical User Interface

In an effort to simplify the creation of application specifications and help visualize the status of executions running on resources around the world, we

implemented a graphical user interface for Plush called Nebula. In particular, we designed Nebula (as shown in Figures 2a, 2b, and 3) to simplify the process of specifying and managing applications running across the PlanetLab wide-area testbed. Plush obtains data from the PlanetLab Central (PLC) database to determine what hosts a user has access to, and Nebula uses this information to plot the sites on the map. To start using Nebula, users have the option of building their Plush application specification from scratch or loading a preexisting XML document representing an application specification. Upon loading the application specification, the user runs the application by clicking the Run button from the Plush toolbar, which causes Plush to start locating and acquiring resources.

The main Nebula window contains four tabs that show different information about user's application. In the "World View" tab, users see an image of a world map with colored dots indicating PlanetLab hosts. Different colored dots on the map indicate sites involved in the current application. In Figure 2a, the dots (which range from red to green on a user's screen) show PlanetLab sites involved in the current application. The grey dots are other available PlanetLab sites that are not currently being used by Plush. As the application proceeds through the different phases of execution, the sites change color, allowing the user to visualize the progress of their application. When failures occur, the impacted sites turn red, giving the user an immediate visual indication of the problem. Similarly, green dots indicate that the application is executing correctly. If a user wishes to establish an SSH connection directly to a particular resource, they can simply right-click on a host in the map and choose the SSH option from the pop-up menu. This opens a new tab in Nebula containing an SSH terminal to the host. Users can also mark hosts as failed by right-clicking and choosing the Fail option from the pop-up menu if they are able to determine failure more quickly than Plush's automated techniques. Failed hosts are completely removed from the execution.

Users retrieve more detailed usage statistics and monitoring information about specific hosts (such as CPU load, free memory, or bandwidth usage) by double clicking on the individual sites in the map. This opens a second window that displays real-time graphs based on data retrieved from resource monitoring tools, as shown in the bottom right corner of Figure 2a. The second smaller window displays a graph of the CPU or memory usage, and the status of the application on each host. Plush currently provides built-in support for monitoring CoMon [25] data on PlanetLab machines, which is the source of the CPU and memory data. Additionally, if the user wishes to view the CPU usage or percentage of free memory available across all hosts, there is a menu item under the PlanetLab menu that changes the colors of the dots on the map such that red means high CPU usage or low free

memory, and green indicates low CPU usage and high free memory. Users can also add and remove hosts to their PlanetLab account (or *slice* in PlanetLab terminology) directly by highlighting regions of the map and choosing the appropriate menu option from the PlanetLab menu. Additionally, users can renew their PlanetLab slice from Nebula.

The second tab in the Nebula main window is the “Application View.” The Application View tab, shown in Figure 2b, allows users to build Plush application specifications using the blocks described previously. Alternatively, users may load an existing XML file describing an application specification by choosing the Load Application menu option under the File menu. There is also an option to save a new application specification to an XML file for later use. After creating or loading an application specification, the Run button located on the Application View tab starts the application.

The Plush blocks in the application specification change to green during the execution of the application to indicate progress. After an application begins execution, users have the option to “force” an application to skip ahead to the next phase of execution (which corresponds to releasing a synchronization barrier), or aborting an application to terminate execution across all resources. Once the application aborts or completes execution, the user may either save their application specification, disconnect from the Plush communication mesh, restart the same application, or load and run a new application by choosing the appropriate option from the File menu.

The third tab is the “Resource View” tab. This tab is blank until an application starts running. During execution, this tab lists the specific machines that are currently involved in the application. If failures occur

during execution, the list of machines is updated dynamically, such that the Resource View tab always contains an accurate listing of the machines that are in use. The resources are separated into components, so that the user knows which resources are assigned to which tasks in their application.

The fourth tab in Nebula is called the “Host View” tab, shown in Figure 3. This tab contains a table that displays the hostname of all available resources. In the right column, the status of the host is shown. The purpose of this tab is to give users another alternative to visualize the status of an executing application. The status of the host in the right column corresponds to the color of the dot in the “World View” tab. This tab also allows users to run shell commands simultaneously on several resources, and view the output. As shown in Figure 3, users can select multiple hosts as once, run a command, and the output is shown in the text-box at the bottom of the window. Note that hosts do not have to be involved in an application in order to take advantage of this feature. Plush will connect to any available resources and run commands on behalf of the user. Just as in the World View tab, right-clicking on hosts in the Host View tab opens a pop-up menu that enables users to SSH directly to the hosts.

Command-line Interface

Motivated by the popularity and familiarity of the shell interface in UNIX, Plush further streamlines the develop-deploy-debug cycle for distributed application management through a simple command-line interface where users deploy, run, monitor, and debug their distributed applications running on hundreds of remote machines. Plush combines the functionality of a distributed shell with the power of an application

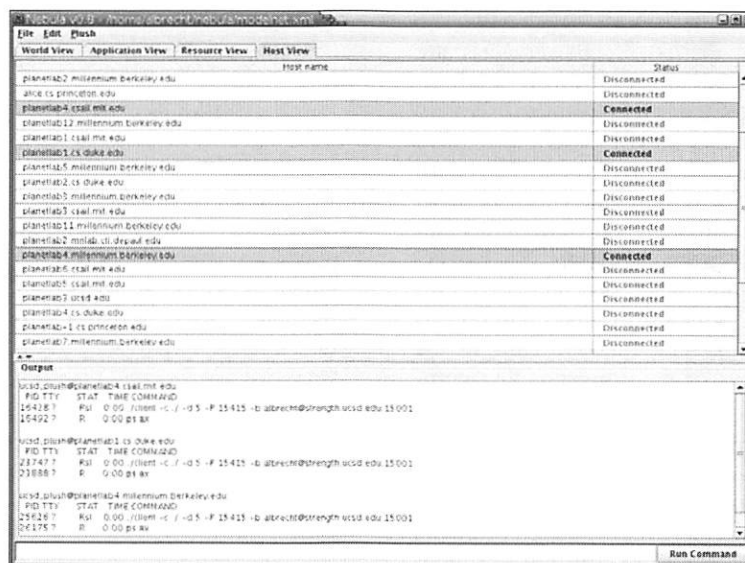


Figure 3: Nebula Host View tab showing PlanetLab resources. This tab allows users to select multiple hosts at once and run shell commands on the selected resources. The text-box at the bottom shows the output from the command.

controller to provide a robust execution environment for users to run their applications. From a user's standpoint, the Plush terminal looks like a shell. Plush supports several commands for monitoring the state of an execution, as well as commands for manipulating the application specification during execution. Table 1 shows some of the available commands.

Command	Description
load <file>	Load application specification
connect <host>	Connect to host and start client
disconnect	Close all connections and clients
info nodes	Print all resource information
info mesh	Print communication fabric status info
info control	Print application control state info
run	Start the application (after load)
shell <cmd>	Run "cmd" on all connected resources

Table 1: Sample Plush terminal commands.

Programmatic Interface

Many commands that are available via the Plush command-line interface are also exported via an XML-RPC interface to deliver similar functionality as the command-line to those who desire programmatic access. This allows Plush to be scripted and used for remote execution and automated application management, and also enables the use of external services for resource discovery, creation, and acquisition within Plush. In addition to the commands that Plush exports, external services and programs may also register themselves with Plush so that the controller can send callbacks to the XML-RPC client when various actions occur during the application's execution.

Figure 4 shows the Plush XML-RPC API. The functions shown in the `PlushXmlRpcServer` class are available to users who wish to access Plush programmatically in scripts, or for external resource discovery and acquisition services that need to add and remove resources from the Plush resource pool. The `plushAddNode(HashMap)` and `plushRemoveNode(string)` calls add and remove nodes from the resource pool, respectively. `setXmlRpcClientUrl(string)` registers XML-RPC clients for callbacks, while `plushTestConnection()` simply tests the connection to the Plush server and returns "Hello World." The remaining function calls in the class mimic the behavior of the corresponding command-line operations.

Aside from resource discovery and acquisition services, the XML-RPC API allows for the implementation of different user interfaces for Plush. Since almost all of the Plush terminal commands are available as XML-RPC function calls, users are free to implement their own customized environment specific user interface without understanding or modifying the internals of the Plush implementation. This is beneficial

because it gives the users more flexibility to develop in the programming language of their choice. Most mainstream programming languages have support for XML-RPC, and hence users are able to develop interfaces for Plush in any language, provided that the chosen language is capable of handling XML-RPC. For example, Nebula is implemented in Java, and uses the XML-RPC interface shown in Figure 4 to interact with a Plush controller. To increase the functionality and simplify the development of these interfaces, the Plush XML-RPC server has the ability to make callbacks to programs that register with the Plush controller via `setXmlRpcClientUrl(string)`. Some of the more common callback functions are shown in the bottom of Figure 4 in class `PlushXmlRpcCallback`. Note that these callbacks are only useful if the programmatic client implements the corresponding functions.

```
class PlushXmlRpcServer extends XmlRpcServer {
    void plushAddNode(HashMap properties);
    void plushRemoveNode(string hostname);
    string plushTestConnection();
    void plushCreateResources();
    void plushLoadApp(string filename);
    void plushRunApp();
    void plushDisconnectApp(string hostname);
    void plushQuit();
    void plushFailHost(string hostname);
    void setXmlRpcClientUrl(string clientUrl);
}

class PlushXmlRpcCallback extends XmlRpcClient {
    void sendPlanetLabSlices();
    void sendSliceNodes(string slice);
    void sendAllPlanetLabNodes();
    void sendApplicationExit();
    void sendHostStatus(string host);
    void sendBlockStatus(string block);
    void sendResourceMatching(HashMap matching);
}
```

Figure 4: Plush XML-RPC API.

Implementation Details

Plush is a publicly available software package consisting of over 60,000 lines of C++ code. Plush depends on several C++ libraries, including those provided by `xmlrpc-c`, `curl`, `xml2`, `zlib`, `math`, `openssl`, `readline`, `curses`, `boost`, and `pthread`s. The command-line interface also depends on packages for `lex` and `yacc` (we use `flex` and `bison`).

In addition to the main C++ codebase, Plush uses several simple perl scripts for interacting with the PlanetLab Central database and bootstrapping resources. Plush runs on most UNIX-based platforms, including Linux, FreeBSD, and Mac OS X, and a single Plush controller can manage clients running on different operating systems. The only prerequisite for using Plush on a resource is the ability to SSH to the resource. Currently Plush is being used to manage applications on

PlanetLab, ModelNet, and Xen virtual machines [5] in our research cluster.

Nebula consists of approximately 25,000 lines of Java code. Nebula communicates with Plush using the XML-RPC interface. XML-RPC is implemented in Nebula using the Apache XML-RPC client and server packages. In addition, Nebula uses the JOGL implementation of the OpenGL graphics package for Java. Nebula runs in any computing environment that supports Java, including Windows, Linux, FreeBSD, and Mac OS X among others. Note that since Nebula and Plush communicate solely via XML-RPC, it is not necessary to run Nebula on the same physical machine as the Plush controller.

Usage Scenarios

One of the primary goals of our work is to build a generic application management framework that supports execution in any execution environment. This is mainly accomplished through the Plush *resource* abstraction. In Plush, resources are computing devices capable of hosting applications, such as physical machines, emulated hosts, or virtual machines. To show

that Plush achieves this goal, in this section we take a closer look at specific uses of Plush in different distributed computing environments, including a live deployment testbed, an emulated network, and a cluster of virtual machines.

PlanetLab Live Deployment

To demonstrate Plush's ability to manage the live deployment of applications, we revisit our previous example from the second section and show how Plush manages SWORD [23] on PlanetLab. Recall that SWORD is a resource discovery service that relies on host monitors running on each PlanetLab machine to report information periodically about their resource usage. This data is stored in a DHT (distributed hash table), and later accessed by SWORD clients to respond to requests for groups of resources that have specific characteristics. SWORD is a service that helps PlanetLab users find the best set of resources based on the priorities and requirements specified, and is an example of a long-running Internet service.

The XML application specification for SWORD is shown in Figure 5. Note that this specification could be built using Nebula, in which case the user would

```
<?xml_version="1.0" encoding="utf"-8?>
<plush>
  <project_name="sword">
    <software_name="sword_software" type="tar">
      <package_name="sword.tar" type="web">
        <path>http://plush.ucsd.edu/sword.tar</path>
        <dest>sword.tar</dest>
      </package>
    </software>
    <component_name="sword_participants">
      <rspec>
        <num_hosts_min="10" max="800"/>
      </rspec>
      <resources>
        <resource_type="planetlab" group="ucsd_sword"/>
      </resources>
      <software_name="sword_software"/>
    </component>
    <application_block_name="sword_app_block" service="1"
      reconnect_interval="300">
      <execution>
        <component_block_name="participants">
          <component_name="sword_participants"/>
          <process_block_name="sword">
            <process_name="sword_run">
              <path>dd/planetlab/run sword</path>
            </process>
          </process_block>
        </component_block>
      </execution>
    </application_block>
  </project>
</plush>
```

Figure 5: SWORD application specification.

never have to edit the XML directly. The top half of the specification in Figure 5 defines the SWORD software package and the component (resource group) required for the application. Notice that SWORD uses one component consisting of hosts assigned to the `ucsd_sword` PlanetLab slice.

An interesting feature of this component definition is the `num_hosts` tag. Since SWORD is a service that wants to run on as many nodes as possible, a range of acceptable values is used rather than a single number. In this case, as long as 10 hosts are available, Plush will continue managing SWORD. Since the max is set to 800, Plush will not look for more than 800 resources to host SWORD. Since PlanetLab contains less than 800 hosts, this means that SWORD will attempt to run on all PlanetLab resources.

The lower half of the application specification defines the application block, component block, and process block that describes the SWORD execution. The application block contains a few key features that help Plush react to failures more efficiently for long-running services. When defining the application block object for SWORD, we include special `“service”` and `“reconnect_interval”` attributes. The service attribute tells the Plush controller that SWORD is a long-running service and requires different default behaviors for initialization and failure recovery. For example, during application initialization the controller does not wait for all participants to install the software before starting all hosts simultaneously. Instead, the controller instructs individual clients to start the application as soon as they finish installing the software, since there is no reason to synchronize the execution across all hosts. Further, if a process fails when the service attribute has been specified, the controller attempts to restart SWORD on that host without aborting the entire application.

The `reconnect_interval` specifies the period of time the controller waits before rerunning the resource discovery and acquisition unit. For long running services, hosts often fail and recover during execution. The `reconnect_interval` attribute tells the controller to check for new hosts that have come alive since the last run of the resource discovery unit. The controller also unsets any hosts that had previously been marked as `“failed”` at this time. This is the controller’s way of `“refreshing”` the list of available hosts. The controller continues to search for new hosts until reaching the maximum `num_hosts` value, which is 800 in our case.

Evaluating Fault Tolerance

To demonstrate Plush’s ability to automatically recover from host failures for long running services, we ran SWORD on PlanetLab with 100 randomly chosen hosts, as shown in Figure 6. The host set includes machines behind DSL links as well as hosts from other continents. When Plush starts the application, the controller starts the Plush client on 100 randomly chosen

PlanetLab machines, and they each begin downloading the SWORD software package (38 MB).

It takes approximately 1000 seconds for all hosts to successfully download, install, and start SWORD. At time $t = 1250s$, we kill the SWORD process on 20 randomly chosen hosts to simulate host failure. Normally, Plush would automatically try to restart the SWORD process on these hosts. However, we disabled this feature to simulate host failures and force a rematching. The remote Plush clients notify the controller that the hosts have failed, and the controller begins to find replacements for the failed machines. The replacement hosts join the Plush overlay and start downloading the SWORD software. As before, Plush chooses the replacements randomly, and low bandwidth/high latency links have a great impact on the time it takes to fully recover from the host failure. At $t = 2200s$, the service is restored on 100 machines.

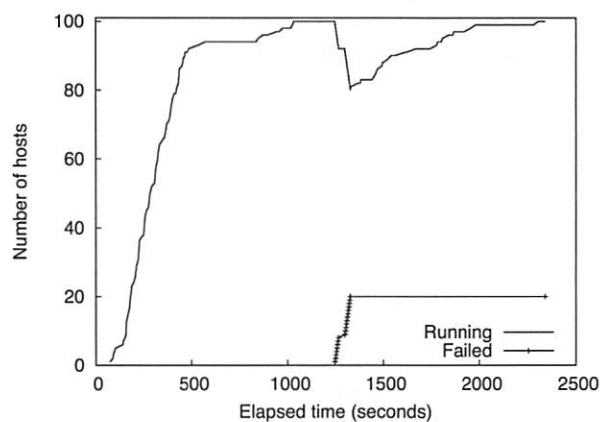


Figure 6: SWORD running on 100 randomly chosen PlanetLab hosts. At $t=1250$ seconds, we fail 20 hosts. The Plush controller finds new hosts, who start the Plush client process and begin downloading and installing the SWORD software. Service is fully restored at approximately $t=2200$ seconds.

Using Plush to manage long-running services like SWORD alleviates the burden of manually probing for failures and configuring/reconfiguring hosts. Further, Plush interfaces directly with the PlanetLab Central API, which means that users can automatically add hosts to their slice and renew their slice using Plush. This is beneficial since services typically want to run on as many PlanetLab hosts as possible, including any new hosts that come online. In addition, Plush simplifies the task of debugging problems by providing a single point of control for all connected PlanetLab hosts. Thus, if a user wants to view the memory consumption of their service across all connected hosts, a single Plush command retrieves this information, making it easier to maintain and monitor a service running on hundreds of resources scattered around the world.

ModelNet Emulation

Aside from PlanetLab resources, Plush also supports running applications on virtual hosts in emulated

environments. In this section we discuss how Plush supports using ModelNet [29] emulated resources to host applications. In addition, we will discuss how a batch scheduler uses the Plush programmatic interface to perform remote job execution.

Mission is a simple batch scheduler used to manage the execution of jobs that run on ModelNet in our research cluster. ModelNet is a network emulation environment that consists of one or more Linux edge nodes and a set of FreeBSD core machines running a specialized ModelNet kernel. The code running on the edge hosts routes packets through the core machines, where the packets are subjected to the delay, bandwidth, and loss specified in a target topology. A single physical machine hosts multiple "virtual" IP addresses that act as emulated resources on the Linux edge hosts.

To setup the ModelNet computing environment with the target topology, two phases of execution are required: deploy and run. Before running any applications, the user must first *deploy* the desired topology on each physical machine, including the FreeBSD core. The deploy process essentially instantiates the emulated hosts, and installs the desired topology on all machines. Then, after setting a few environment variables, the user is free to *run* applications on the emulated hosts using virtual IP addresses just as applications are run on physical machines using real IP addresses.

A single ModelNet experiment typically consumes almost all of the computing resources available on the physical machines involved. Thus, when running an experiment, it is essential to restrict access to the machines so that only one experiment is running at a time. Further, there are a limited number of FreeBSD core machines running the ModelNet kernel available, and access to these hosts must also be arbitrated. Mission is a batch scheduler developed locally to help accomplish this goal by allowing the resources to be efficiently shared among multiple users. ModelNet users submit their jobs to the Mission job queue, and as the machines become available, Mission pulls jobs off the queue and runs them on behalf of the user, ensuring that no two jobs are run simultaneously.

A Mission job submission has two components: a Plush application specification and resource directory file. For ModelNet, the directory file contains information about both the physical and virtual (emulated) resources on which the ModelNet experiment will run. In the resource directory file, some entries include two extra parameters, "vip" and "vn", which define the virtual IP address and virtual number (similar to a hostname) for the emulated resources. In addition to the directory file that is used to populate the Plush resource pool, users also submit an application specification describing the application they wish to run on the emulated topology to the Mission server.

The application specification submitted to Mission contains two component blocks separated by a

synchronization barrier. The first component block describes the processes that run on the physical machines during the deployment phase (where the emulated topology is instantiated). The second component block defines the processes associated with the target application. When the controller starts the Plush clients on the emulated hosts, it specifies extra command line arguments that are defined in the directory file by the "vip" and "vn" attributes. This sets the appropriate ModelNet environment variables, ensuring that all commands run on that client on behalf of the user inherit those settings.

When a user submits a Plush application specification and directory file to Mission, the Mission server parses the directory file to identify which resources are needed to host the application. When those resources become available for use, Mission starts a Plush controller on behalf of the user using the Plush XML-RPC interface. Mission passes Plush the directory file and application specification, and continues to interact throughout the execution of the application via XML-RPC. After Plush notifies Mission that the execution has ended, Mission kills the Plush process and reports back to the user with the results. Any terminal output that is generated is emailed to the user.

Plush jobs are currently being submitted to Mission on a daily basis at UCSD. These jobs include experimental content distribution protocols, distributed model checking systems, and other distributed applications of varying complexity. Mission users benefit from Plush's automated execution capabilities. Users simply submit their jobs to Mission and receive an email when their task is complete. They do not have to spend time configuring their environment or starting the execution. Individual host errors that occur during execution are aggregated into one message and returned back to the user in the email. Logfiles are collected in a public directory on a common file system and labeled with a job ID, so that users are free to inspect the output from individual hosts if desired.

Virtual Machine Deployment

In all of the examples discussed above, the pool of resources available to Plush is known at startup. In the PlanetLab examples, Plush uses slice information to determine the set of user-accessible hosts. For ModelNet, the emulated topology includes specific information about the virtual hosts to be created and this information is passed to Plush in the directory file. We next describe how Plush manages applications in environments without fixed sets of machines, but rather underlying capabilities to create and destroy resources on demand.

Shirako [16] is a utility computing framework. Through programmatic interfaces, Shirako allows users to create dynamic on-demand clusters of resources, including storage, network paths, physical servers, and virtual machines. Shirako is based on a resource leasing

abstraction, enabling users to negotiate access to resources. Usher [21] is a virtual machine scheduling system for cluster environments. It allows users to create their own virtual machines or clusters. When a user requests a virtual machine, Usher uses data collected by virtual machine monitors to make informed decisions about when and where the virtual machine should run.

We have extended Plush to interface with both Shirako and Usher. Through its XML-RPC interface, Plush interacts with the Shirako and Usher servers. As resources are created and destroyed, the resource pool in Plush is updated to include the current set of leased resources. Using this dynamic resource pool, Plush manages applications running on potentially temporary virtual machines in the same way that applications are managed in static environments like PlanetLab. Thus, using the resource abstractions provided by Plush, users are able to run their applications on PlanetLab, ModelNet, or on clusters of virtual machines without ever having to worry about the underlying details of the environment.

To support dynamic resource creation and management, we augment the Plush application specification with a description of the desired virtual machines as shown in Figure 7. Specifically, the Plush application specification needs to include information about the desired attributes of the resources so that this information can be passed on to either Shirako or Usher. Shirako and Usher currently create Xen [5] virtual machines (as indicated by the “type” flag in the resource description) with the CPU speed, memory, disk space, and maximum bandwidth specified in the resource request. As the Plush controller parses the application

specification, it stores the resource description. Then when the create resource command is issued either via the terminal interface or programmatically through XML-RPC, Plush contacts the appropriate Shirako or Usher server and submits the resource request. Once the resources are ready for use, Plush is informed via an XML-RPC callback that also contains contact information about the new resources. This callback updates the Plush resource pool and the user is free to start applications on the new resources by issuing the run command to the Plush controller.

Though the integration of Plush and Usher is still in its early stages, Plush is being used by Shirako users regularly at Duke University. While Shirako multiplexes resources on behalf of users, it does not provide any abstractions or functionality for using the resources once they have been created. On the other hand, Plush provides abstractions for managing distributed applications on remote machines, but provides no support for multiplexing resources. A “resource” is merely an abstraction in Plush to describe a machine (physical or virtual) that can host a distributed application. Resources can be added and removed from the application’s resource pool, but Plush relies on external mechanisms (like Shirako and Usher) for the creation and destruction of resources.

The integration of Shirako and Plush allows users to seamlessly leverage the functionality of both systems. While Shirako provides a web interface for creating and destroying resources, it does not provide an interface for using the new resources, so Shirako users benefit from the interactivity provided by the Plush shell. Researchers at Duke are currently using

```
<?xml_version="1.0" encoding="utf-8"?>
<plush>
  <project_name="simple">
    <component_name="Group1">
      <rspec>
        <num_hosts>10</num_hosts>
        <shirako>
          <num_hosts>10</num_hosts>
          <type>1</type>
          <memory>200</memory>
          <bandwidth>200</bandwidth>
          <cpu>50</cpu>
          <lease_length>600</lease_length>
          <server>http://shirako.cs.duke.edu:20000</server>
        </shirako>
      </rspec>
      <resources>
        <resource_type="ssh" group="shirako"/>
      </resources>
    </component>
  </project>
</plush>
```

Figure 7: Plush component definition containing Shirako resources. The resource description contains a lease parameter which tells Shirako how long the user needs the resources.

Plush to orchestrate workflows of batch tasks and perform data staging for scientific applications including BLAST [3] on virtual machine clusters managed by Shirako [14].

Related Work

The functionality that Plush provides is related to work in a variety of areas. With respect to remote job execution, there are several tools available that provide a subset of the features that Plush supports, including cfengine [9], gexec [10], and vxargs [20]. The difference between Plush and these tools is that Plush provides more than just remote job execution. Plush also supports mechanisms for failure recovery, and automatic reconfiguration due to changing conditions. In general, the pluggable aspect of Plush allows for the use of existing tools for actions like resource discovery and allocation, which provides more advanced functionality than most remote job execution tools.

From the user's point of view, the Plush command-line is similar to distributed shell systems such as GridShell [31] and GCEShell [22]. These tools provide a user-friendly language abstraction layer that support script processing. Both tools are designed to work in Grid environments. Plush provides a similar functionality as GridShell and GCEShell, but unlike these tools, Plush works in a variety of environments.

In addition to remote job execution tools and distributed shells, projects like the PlanetLab Application Manager (appmanager) [15] and SmartFrog [13] focus specifically on managing distributed applications. appmanager is a tool for maintaining long running services and does not provide support for short-lived executions. SmartFrog [13] is a framework for describing, deploying, and controlling distributed applications. It consists of a collection of daemons that manage distributed applications and a description language to describe the applications. Unlike Plush, SmartFrog is a not a turnkey solution, but rather a framework for building configurable systems. Applications must adhere to a specific API to take advantage of SmartFrog's features.

There are also several commercially available products that perform functions similar to Plush. Namely, Opsware [24] and Appistry [4] provide software solutions for distributed application management. Opsware System 6 allows customers to visualize many aspects of their systems, and automates software management of complex, multi-tiered applications. The Appistry Enterprise Application Fabric strives to deliver application scalability, dependability, and manageability in grid computing environments. In comparison to Plush, both of these tools focus more on enterprise application versioning and package management, and less on providing support for interacting with experimental distributed systems.

The Grid community has several application management projects with goals similar to Plush, including

Condor [8] and GrADS/vGrADS [7]. Condor is a workload management system for compute-intensive jobs that is designed to deploy and manage distributed executions. Where Plush is designed to deploy and manage naturally distributed tasks with resources spread across several sites, Condor is optimized for leveraging underutilized cycles in desktop machines within an organization where each job is parallelizable and compute-bound. GrADS/vGrADS [7] provides a set of programming tools and an execution environment for easing program development in computational grids. GrADS focuses specifically on applications where resource requirements change during execution. The task deployment process in GrADS is similar to Plush. Once the application starts execution, GrADS maintains resource requirements for compute intensive scientific applications through a stop/migrate/restart cycle. Plush, on the other hand, supports a far broader range of recovery actions.

Within the realm of workflow management, there are tools that provide more advanced functionality than Plush. For example, GridFlow [11], Kepler [19], and the other tools described in [32] are designed for advanced workflow management in Grid environments. The main difference between these tools and Plush is that they focus solely on workflow management schemes. Thus they are not well suited for managing applications that do not contain workflows, such as long-running services.

Lastly, the Globus Toolkit [12] is a framework for building Grid systems and applications, and is perhaps the most widely used software package for Grid development. Some components of Globus provide a similar functionality as Plush. With respect to our application specification language, the Globus Resource Specification Language (RSL) provides an abstract language for describing resources that is similar in design to our language. The Globus Resource Allocation Manager (GRAM) processes requests for resources, allocates the resources, and manages active jobs in Grid environments, providing much of the same functionality as Plush does. The biggest difference between Plush and Globus is that Plush provides a user-friendly shell interface where users directly interact with their applications. Globus, on the other hand, is a framework, and each application must use the APIs to create the desired functionality. In the future, we plan to integrate Plush with some of the Globus tools, such as GRAM and RSL. In this scenario Plush will act as a front-end user interface for the tools available in Globus.

Conclusion

Plush is an extensible application control infrastructure designed to meet the demands of a variety of distributed applications. Plush provides abstractions for resource discovery and acquisition, software installation, process execution, and failure recovery in distributed environments. When an error is detected,

Plush has the ability to perform several application-specific actions, including restarting the computation, finding a new set of resources, or attempting to adapt the application to continue execution and maintain liveness. In addition, Plush provides new relaxed synchronization primitives that help applications achieve good throughput even in unpredictable wide-area conditions where traditional synchronization primitives are too strict to be effective.

Plush is in daily use by researchers worldwide, and user feedback has been largely positive. Most users find Plush to be an "extremely useful tool" that provides a user-friendly interface to a powerful and adaptable application control infrastructure. Other users claim that Plush is "flexible enough to work across many administrative domains (something that typical scripts do not do)." Further, unlike many related tools, Plush does not require applications to adhere to a specific API, making it easy to run distributed applications in a variety of environments. Our users tell us that Plush is "fairly easy to get installed and setup on a new machine. The structure of the application specification largely makes sense and is easy to modify and adapt."

Although Plush has been in development for over three years now, we still have some features that need improvement. One important area for future enhancements is error reporting. Debugging applications is inherently difficult in distributed environments. We try to make it easier for researchers using Plush to locate and diagnose errors, but this is a difficult task. For example, one user says that "when things go wrong with the experiment, it's often difficult to figure out what happened. The debug output occasionally does not include enough information to find the source of the problem." We are currently investigating ways to allow application specific error reporting, and ultimately simplify the task of debugging distributed applications in volatile environments.

Author Biographies

Jeannie Albrecht is an Assistant Professor of Computer Science at Williams College in Williamstown, Massachusetts. She received her Ph.D. in Computer Science from the University of California, San Diego in June 2007 under the supervision of Amin Vahdat and Alex C. Snoeren.

Ryan Braud is a fourth-year doctoral student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science and Mathematics from the University of Maryland, College Park in 2004.

Darren Dao is a graduate student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science from the University of California, San Diego in 2006.

Nikolay Topilski is a graduate student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science from the University of California, San Diego in 2002.

Christopher Tuttle is a Software Engineer at Google in Mountain View, California. He received his M.S. in Computer Science from the University of California, San Diego in December 2005 under the supervision of Alex C. Snoeren.

Alex C. Snoeren is an Assistant Professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received his Ph.D. in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 2003 under the supervision of Hari Balakrishnan and M. Frans Kaashoek.

Amin Vahdat is a Professor in the Department of Computer Science and Engineering and the Director of the Center for Networked Systems at the University of California, San Diego. He received his Ph.D. in Computer Science from the University of California, Berkeley in 1998 under the supervision of Thomas Anderson. Before joining UCSD in January 2004, he was on the faculty at Duke University from 1999-2003.

Bibliography

- [1] Albrecht, J., C. Tuttle, A. C. Snoeren, and A. Vahdat, "Loose Synchronization for Large-Scale Networked Systems," *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.
- [2] Albrecht, J., C. Tuttle, A. C. Snoeren, and A. Vahdat, "PlanetLab Application Management Using Plush," *ACM Operating Systems Review (OSR)*, Vol. 40, Num. 1, 2006.
- [3] Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, Vol. 215, 1990.
- [4] Appistry, <http://www.appistry.com/>.
- [5] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.
- [6] Bavier, A., M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating Systems Support for Planetary-Scale Network Services," *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
- [7] Berman, F., H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D.

- Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan, "New Grid Scheduling and Rescheduling Methods in the GrADS Project," *International Journal of Parallel Programming (IJPP)*, Vol. 33, Num. 2-3, 2005.
- [8] Bricker, A., M. Litzkow, and M. Livny, "Condor Technical Summary," *Technical Report 1069*, University of Wisconsin-Madison, CS Department, 1991.
- [9] Burgess, M., "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 3, 1995.
- [10] Chun, B., *gexec*, <http://www.theether.org/gexec/>.
- [11] Coa, J., S. Jarvis, S. Saini, and G. Nudd, "GridFlow: Workflow Management for Grid Computing," *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CC-Grid)*, 2003.
- [12] Foster, I., *A Globus Toolkit Primer*, 2005, http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.
- [13] Goldsack, P., J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft, "SmartFrog: Configuration and Automatic Ignition of Distributed Applications," *HP Openview University Association Conference (HP OVUA)*, 2003.
- [14] Grit, L., D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht, "Harnessing Virtual Machine Resource Control for Job Management," *Proceedings of the Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.
- [15] Huebsch, R., *PlanetLab Application Manager*, <http://appmanager.berkeley.intel-research.net>.
- [16] Irwin, D., J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum, "Sharing Networked Resources with Brokered Leases," *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.
- [17] Jordan, H. F., "A Special Purpose Architecture for Finite Element Analysis," *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1978.
- [18] Ludtke, S., P. Baldwin, and W. Chiu, "EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions," *Journal of Structural Biology*, Vol. 122, 1999.
- [19] Ludäscher, B., I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows (CC: P&E)*, Vol. 18, Num. 10, 2005.
- [20] Mao, Y., *vxargs*, <http://dharma.cis.upenn.edu/planetlab/vxargs/>.
- [21] McNett, M., D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An Extensible Framework for Managing Clusters of Virtual Machines," *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2007.
- [22] Nacar, M. A., M. Pierce, and G. C. Fox, "Developing a Secure Grid Computing Environment Shell Engine: Containers and Services," *Neural, Parallel, and Scientific Computations (NPSC)*, Vol. 12, 2004.
- [23] Oppenheimer, D., J. Albrecht, D. Patterson, and A. Vahdat, "Design and Implementation Trade-offs for Wide-Area Resource Discovery," *Proceedings of the IEEE Symposium on High Performance Distributed Computing (HPDC)*, 2005.
- [24] Opsware, <http://www.opsware.com/>.
- [25] Park, K. and V. S. Pai, "CoMon: A Mostly-Scalable Monitoring System for PlanetLab," *ACM Operating Systems Review (OSR)*, Vol. 40, Num. 1, 2006.
- [26] Peterson, L., T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," *Proceedings of the ACM Workshop on Hot Topics in Networks (Hot-Nets)*, 2002.
- [27] Plush, <http://plush.ucsd.edu>.
- [28] Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System," *Communications of the Association for Computing Machinery (CACM)*, Vol. 17, Num. 7, 1974.
- [29] Vahdat, A., K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and Accuracy in a Large-Scale Network Emulator," *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.
- [30] Waldspurger, C. A., "Memory Resource Management in VMware ESX Server," *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.
- [31] Walker, E., T. Minyard, and J. Boisseau, "GridShell: A Login Shell for Orchestrating and Coordinating Applications in a Grid Enabled Environment," *Proceedings of the International Conference on Computing, Communications and Control Technologies (CCCT)*, 2004.
- [32] Yu, J. and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing," *Journal of Grid Computing (JGC)*, Vol. 3, Num. 3-4, 2005.

Everlab – A Production Platform for Research in Network Experimentation and Computation

Elliot Jaffe, Danny Bickson, and Scott Kirkpatrick – Hebrew University of Jerusalem, Israel

ABSTRACT

We have pioneered the deployment of EverLab, a production level private PlanetLab system using high-end clusters spread over Europe. EverLab supports both experimentation and computational work, incorporating many of the features found on Grid systems. This paper describes the decision process that led us to choose PlanetLab and the challenges that we faced during our implementation and production phases. We detail the monitoring systems that were deployed on EverLab and their impact on our management policies. The paper concludes with suggestions for future work on private PlanetLabs and federated systems.

Introduction

Evergrow is a European Commission Sixth Framework Integrated Project with around 28 participating research organizations spread across Europe and the Middle East. The project combines research efforts including network measurement [11, 17], distributed systems [1], and complex systems research [9]. Our researchers span the range from systems developers to physicists. At the onset of the project, we realized a need to provide computational and experimental tools for our members. We purchased a set of eight IBM HS20 clusters co-located with some of our research members. Each cluster has 16 blades, where one blade is a storage server and another blade is used for configuration management. We allocated support expenses to the hosting members in order to provide administration, maintenance and support to our users.

Intentionally, the clusters were spread across eight European facilities: Aston University – UK, Université Paris-Sud 11 “Orsay” (UPSXI) – France, Istituto Nazionale per la Fisica della Materia (INFN) – Rome, Italy, Collegium Budapest Egyesület (COLBUD) – Hungary, Tel Aviv University (TAU) – Israel, Otto-von-Guericke-Universität at Magdeburg (UNI MD) – Germany, Université Catholique de Louvain (UCL) – Belgium and Swedish Institute of Computer Science (SICS) – Sweden. The clusters were deployed in different sites so that we could run “real-world” network experiments using existing wide area network links.

In September of 2004, the team met to decide how to integrate the clusters into a shared resource. It was agreed to setup a VPN between the clusters with a master LDAP server for authentication. We intended to use IBM’s GPFS file system to make our storage available throughout the cluster.

For various reasons, our agreed upon approach was not implemented. The reasons included technical problems, networking policies and the availability of

local resources. One year later in September of 2005, we undertook a survey of our clusters. We found a



Figure 1: Geographic location of EverLab clusters.

very sad state of affairs. Each local administrator had chosen a different stand-alone implementation for their cluster. Initially the clusters had RedHat EL2 installed. Some of the local system administrators changed the operating system to Fedora Core 4, Debian, Ubuntu or Mosix. Many of the clusters were inaccessible because the local network policy forbade open access to “internal” computational resources. At one point, we had eight different operating systems. No part of the original plan was universally implemented and hence our researchers had to request permission from each administrator both for a login id and for a firewall exemption so that they could access the nodes.

Our first task was to deploy a monitoring system for all nodes in the eight clusters. We found that Ganglia [10] was easy to install and required only a small number of changes to existing network policies. Ganglia gave us our first view into what the clusters were doing. The results were disappointing. Many of the nodes were idle.

Having realized that our current approach was not working, we started looking at alternatives. One option was to force all the administrators to adopt a standard platform. This was rejected because each domain had their “standard” platform, be it RedHat, Fedora Core, Debian or BSD. The administrators did not want to be responsible for an unfamiliar system. The other option was to find a standard platform that could be administered centrally, relieving the local administrators from direct interaction with the installed operating system. Two options were suggested: Grid and PlanetLab. A comparison of these two approaches can be found in [21].

We explored using a Grid infrastructure [3]. Grid systems are designed for computation and could have been deployed across our clusters. Grid environments are reasonably well developed. Such a system would have provided a unified login, the ability to deploy applications across the nodes and a strong monitoring and management infrastructure. The problem was that the Grid is optimized for computation. Applications are automatically deployed to available nodes. A large fraction of our researchers wanted to perform experiments where the location of a process is important. When debugging network experiments, it is important to be able to run test scripts and tracing programs directly on each remote node. The Grid infrastructure does not allow this kind of access. Grid computation nodes are accessible to users only through the Grid management system.

At the time PlanetLab [18] had been deployed to around 500 nodes across the world. PlanetLab supports network experimentation across remote distributed nodes. The system is centrally managed from Princeton University in the United States. PlanetLab itself was not a complete solution for two reasons; first, PlanetLab is designed only for network experimentation. A significant fraction of Evergrow researchers needed computational resources. Secondly, at the time, there were no production level Private PlanetLab installations. At October 2005, we initiated a European PlanetLab workshop in EPFL, Switzerland [16]. We found out there was significant interest at both educational institutions and in industry for implementing and using Private PlanetLabs to share and manage remote resources.

In December 2005, we took up the challenge of implementing PlanetLab on the Evergrow clusters. We called the new system EverLab. The path was treacherous. At the time, the PlanetLab software was not

designed for ease of installation. It was a moving target with components being rewritten and upgraded on a regular and unannounced basis. PlanetLab was designed for “low end” computers. It ran on single processor from the Pentium family with 512 MB or RAM, a CDROM and 50 GB of local disk and direct connect keyboard connected to a central USB BUS. Our cluster blades have dual 3 Ghz Xeon processors with 4 GB of local ram, 80 GB of disk, no CDROM and a USB keyboard. It took many months to identify the problems and build the appropriately modified kernels and support files.

We succeeded in deploying a Private PlanetLab system. The system provided a centrally managed platform that was usable for experimentation. We installed Ganglia on EverLab so that we could monitor both the old and new systems from one platform. We also developed a custom-built resource reporting system called EverStats. Together, these tools allow our administrators to track system usage and to identify network and hardware problems. Ganglia told us which machines were in use and EverStats told us who used our system and how each node was allocated.

At this point, EverLab was operational and usable by all Evergrow researchers, but it did not yet support High Performance Computation (HPC). To this end, we deployed the Condor [19] system from the University of Wisconsin. We deployed the Message Passing Interface (MPICH2) [5] on top of Condor. With these two additions, Everlab now supports both computation and networking research.

As far as we know, we were the first production-quality Private PlanetLab. Unlike the original PlanetLab network which is mainly based on regular PC computers, our network is based on high-end servers with Gigabit Internet connection.

EverLab runs on a subset of the 112 EverGrow nodes. It currently includes more than 50 blades in six clusters. All Evergrow researchers can create an account on EverLab and have quick access to the resources without negotiating with eight separate administrative domains. EverLab is monitored 24x7, and problems are quickly identified. The EverLab administrators can handle system level issues. Local administrators respond to hardware related problems.

The rest of this paper describes the challenges and solutions that we encountered during this journey. We detail the value of Ganglia and EverStats to our administration efforts. There are still many opportunities to improve and extend EverLab some of which are described in the Future Work section. Finally, we conclude with our lessons and opinions about the use of EverLab type systems for new research projects.

PlanetLab

Everlab is based on PlanetLab version 3.2 [18]. This section describes the PlanetLab implementation.

Overall, we found PlanetLab to be a very stable platform once the installation process and initial settings were completed.

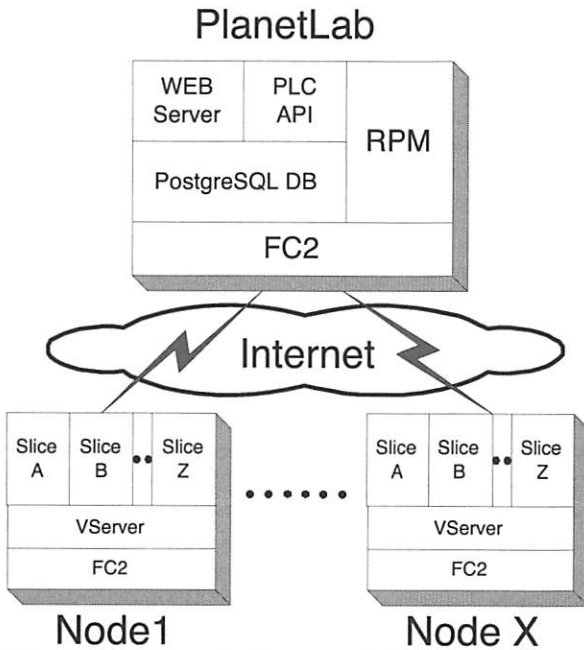


Figure 2: Schematic diagram of the PlanetLab network components.

PlanetLab is a centrally managed collection of distributed computers which are called nodes. The system is designed to be used on a publicly accessible network where all nodes can at a minimum access the central management node. The central management node or PlanetLab Central (PLC) supplies three functions: database for storing system state, web interface for management and a RPM [4] repository for updating the remote nodes. The web interface provides a human readable interface and an XMLRPC interface called PLCAPI for internal use. Remote nodes communicate with the PLC through HTTPS and PLCAPI calls.

At the time of our initial efforts, The PlanetLab Central node ran on Fedora Core 2 (FC2). The PLC used a PostgreSQL database and an Apache web server. Most of PlanetLab was implemented in a mixture of shell and python. The use of common open-source components was a significant factor in our decision to implement PlanetLab. We felt that we could understand, maintain and modify any or all of the components as needed.

Each PlanetLab Version 3 node consists a modified FC2 system. All user activity is performed using virtualization technology implemented by the vserver [8] kernel extension. Node installation is intentionally kept as simple and minimal as possible both to reduce complexity and to increase security. PlanetLab's virtualization unit is called a *slice*. Each slice is a minimal FC2 installation. A user logged into a slice has slice level

superuser privileges through the sudo command. Slices provide compartmentalization between users and system components, thus reducing or eliminating the possibility of one user modifying or removing a file or component necessary to another user or process.

PlanetLab Security

PlanetLab was designed from the outset as a platform for network experimentation. PlanetLab nodes need free access to and from the Internet in order to provide the broadest possible research opportunities and to limit unexpected network interactions caused by firewalls or local network policies. This focus impacted many of the installation, administration and security aspects of PlanetLab.

PlanetLab utilizes asymmetric encryption keys to create secure authenticated communication channels. These keys are used to identify nodes, servers, and users within the system. There are unique keys for run-time and debug mode operations.

All nodes are assumed to be at risk. Even with the strong compartmentalization provided by the vserver slices, in principle an attacker could enter the root domain and take over the machine. To minimize this exposure and to provide a recovery mechanism from a possible penetration, PlanetLab initially boots from a CDROM. The CDROM contacts the PLC and can either enter a debug mode, boot to the exiting disk based kernel, or re-install the node.

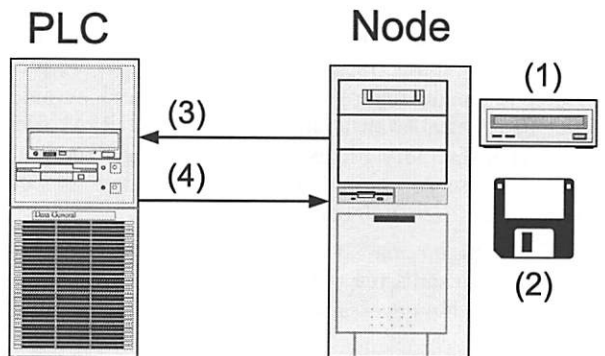


Figure 3: Planetlab boot process. (1) Everlab node boots from CD-ROM. (2) Node gets certificate and identity from floppy drive. (3) Authentication process is done via the PLC. (4) Local files are updated from PLC. (5) Node bootstraps into VServer kernel.

The PlanetLab kernel includes a secure Ping Of Death (POD) implementation which allows the PLC to cause the kernel to reboot given an encrypted secret that only the PLC could have produced.

If a node is suspected of having been compromised, a PlanetLab administrator can cause the node to reboot using the Ping Of Death and to reboot from CDROM into debug mode. At this point, the administrator can log into the node using a special debug

mode SSH key. The administrator can mount the local disk, examine the files and determine if the machine is worth saving. At any point, the administrator can set the nodes status to “reinstall”. On the next reboot, the CDROM based kernel will wipe the disks clean and install a clean system from scratch.

In the 18 months that we have run PlanetLab nodes on the Internet, we have never had a known penetration. We have used both the POD and the Reinstall option to recover from hardware and software errors.

NetFlow: Security Monitoring and Logging

PlanetLab includes a package called NetFlow on each node. The PlanetLab Netflow component is based on the Netfilter [12] ulogd package. This package tracks all network flows, i.e., communications between this node and all other nodes. The data is available over HTTP from port 80 on each node. The flow traces are very useful in determining which slice was responsible for communication to a given node. This can be helpful when debugging an application or experiment. It can also be used when we suspect that the node has been compromised either by an external party or a rogue experiment.

Installation Issues

PL_BOX

The first efforts to deploy Private PlanetLabs were through a package called “pl_box:” PlanetLab in a Box. The package consists of a set of scripts which can download and install all necessary PlanetLab components. The local machine is installed as a PlanetLab Central (PLC) and separate scripts are provided to create deployment CDROM’s and kernels. The PLC installation copies the necessary RPM files to a local directory for use when installing PlanetLab nodes. These RPMs include the FC2 package as well as separate PlanetLab packages.

The pl_box package generates all public and private keys, installs the databases and web applications and creates the necessary cron jobs to keep PlanetLab running.

For our implementation needs, there were two major drawbacks to the pl_box system. First, the installed system is a copy of the PlanetLab system. All the web pages, documents and embedded links point back to the original PlanetLab system instead of the newly installed Private PlanetLab. Secondly, there is no upgrade path for pl_box. Changes made to the original PlanetLab system must be manually imported into the private pl_box. At the time, there was no mechanism for change notification and so the public PlanetLab and the private pl_box system were guaranteed to diverge.

Even with these issues, we have found pl_box to be sufficiently stable for our needs. We have had no serious issues since our deployment more than 18 months ago. The PlanetLab project has since replaced

pl_box with a new system called MyPLC. The My-PLC system offers the ability to customize the user interface for the private installation. It provides a upgrade mechanism through the use of standard RPM source and binary packages available from the PlanetLab development team. There is no formal upgrade path from pl_box to MyPLC and so we will need to re-install our entire system and re-implement our extensions. None-the-less, we believe that MyPLC represents the future of private PlanetLabs and we plan on upgrading to the new system sometime this year.

Our first challenge was to install pl_box. We chose to use a Fedora Core 4 platform for this purpose. At the time, Thierry Parmentelat at Inria in France showed that it was possible to run a PLC on FC4. We decided to install on FC4, given that it was a fresher, more secure release than the default FC2. The installation and production challenges revolved around changes to core packages such as PHP that pl_box required. Debugging these differences gave us our first understanding of the core functionality.

Once the PLC was installed, we moved to installing new nodes. We started with two local machines that had already been PlanetLab nodes. We had no trouble installing these two machines and were now ready to deploy to the clusters. It was here that our real problems began.

Cluster Ownership

We spent many month prototyping and experimenting with PlanetLab to determine its appropriateness for our installations. At the EverGrow general assembly meeting in December 2005, we presented our results and were given the green light to deploy EverLab on the group’s clusters. We then went to each local site administrator and asked for access to deploy the new system. We were met with three types of responses.

Some administrators welcomed us with open arms. We were going to reduce their overhead by managing all of cluster’s software and user issues. These systems were converted to EverLab as soon as the existing researchers had finished their ongoing experiments.

Some site administrators had integrated one or more of their cluster’s nodes into research workflows. These nodes became dedicated to that project and were unavailable to EverLab. At these sites, EverLab was installed on most but not all of the nodes.

Finally, one cluster never made the transition to EverLab. This group had installed a load balancing version of Linux and were able to keep all of the cluster’s node fully loaded more than 90% of the time. It was decided that there was little benefit to be had by moving these nodes to EverLab.

Network Politics

The Evergrow nodes were deployed to eight universities across Europe. Each university had and has

their own network policies. Some of our host universities were already hosting PlanetLab nodes.

We had to fight the network battle at each separate location. PlanetLab minimally requires that:

1. Each node has a public DNS entry.
2. Each node has unhindered access to the PLC.
3. The PLC can send packets to each node for the Ping of Death feature.

In addition, we hoped that all machines would have unhindered access to and from the Internet.

We negotiated with the local system administrators and they negotiated with their local network administrators. In most cases, we were able to get the nodes physically located on a public Internet. The negotiations sometimes required the signatures of university officials or worse, university security officers. All told, it took many months to simply gain access to the remote clusters.

Hardware

The closest cluster to our developers was in Tel-Aviv University (TAU). Unfortunately, TAU has very restrictive access policies and to this day, provides only restricted access to their EverLab nodes. In contrast, The Swedish Institute of Computer Science (SICS) had nodes directly connected to the Internet and were eager to help with our new system. We decided to start testing with the SICS cluster.

Our first challenge was to replace the CDROM based bootstrap process used in PlanetLab nodes. We wanted to maintain the bootstrap ability, but our blades did not have a dedicated CDROM drive or USB drive. We needed to have our nodes boot from the network. We discarded the option of booting from the PLC because of the significant network delays and limited bandwidth between our clusters in Europe and the PLC in Jerusalem. This left the option of booting from a local node.

Each cluster included a management node that was originally designed to provide network boot over DHCP and PXEboot [7]. PXEboot usually downloads a kernel to the target node. The node boots the kernel in diskless mode and uses NFS to mount the root partition from the boot server. We wanted our root partition to be read-only and to be unique for each node. We could have created a separate read-only directory on the management node and mounted it on each boot node. Instead we choose to incorporate the complete root partition into an initrd file. This file is then downloaded to each node as it boots. The EverLab initrd file contains the complete content of the Boot CD. In a standard system, each PlanetLab Boot CD references a diskette which contains the private keys for that machine. PlanetLab allows an administrator to put these keys into the CDROM itself, thus having a custom CDROM for each node.

Our nodes do not have a diskette, thus we needed a custom initrd for each node. We created a generic

initrd and wrote script that copies the generic initrd to a custom file and installs the private keys. This script is then run once for each target node on the local cluster.

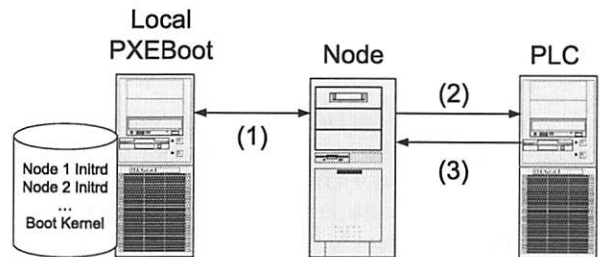


Figure 4 Everlab PXE boot solution. (1) Boot kernel and initrd downloaded from local PXE boot server via PXEboot. (2,3) Authentication process is done via the PLC and local files are updated. Finally, node bootstraps into VServer kernel.

The blades have two Ethernet Network Interface Cards (NICs). We choose to use one NIC for the bootstrap process and the second for the public Internet. We use the private NIC only during the boot process and leave it un-configured during normal EverLab operations. Slices are unable to configure the NICs. The main benefit is that EverLab users have no way to attack or even to see the bootserver. We believe that this increases the probability that the boot image remains intact. Our approach is not as good as a read-only CDROM, but we feel that it strikes the right balance between security and complexity for our system.

Once we had the PXEboot and initrd process working, we were able to boot the default PlanetLab kernel. Unfortunately, our blades were newer than the supported PlanetLab nodes. The running nodes had no network drivers and no keyboard controller. Each blade has a USB keyboard, but the default PlanetLab nodes did not install the appropriate drivers. Similarly, the blades used a network card that was not available when FC2 was first released. We were left with a node that was clearly running something, but that was deaf and blind.

Through trial and error we were able to identify the appropriate drivers and configuration files and to rebuild our initrd files. This process took the better part of a month, but we were finally able to debug and boot our nodes.

Management Challenges

Our intention was to develop a system that could be managed remotely. The PlanetLab system provided most of that functionality. PlanetLab defines four types of capabilities:

1. *Administrators* These users can perform all PlanetLab operations including Ping of Death and re-install. They can enable or disable features and capabilities for other users.
2. *Principal Investigators* These users are responsible for the use of a set of nodes. They can

enable or disable access for their students and can create slices.

3. *User* Can deploy a slice to one or more nodes and can log into those nodes.
4. *Tech* These users can administer their local site, performing admin like functions only on those nodes.

We use the standard PlanetLab definitions, but allocated each of the local system administrators with Principal Investigator and Tech capabilities. These system administrators are then asked to enable accounts for researchers in their institutions. Researchers that do not have a local cluster administrator are managed by the central EverLab administration team.

Hyperthreading Performance Issues

During the decision process, our computation based researchers were concerned that the PlanetLab infrastructure would require a significant fraction of each node's CPU cycles. We were able to show that on our hardware, the difference between a program running on a stock kernel and one running in a slice under PlanetLab was less than 3%. This experiment was not particularly scientific, but it was sufficient to assuage the fears of the HPC researchers.

Our nodes came with two Intel Xeon 3.06 Ghz processors that support Intel's Hyper-threading Technology (HTT) [22]. In theory Hyper-Threading Technology should provide a performance boost of up to 30%. We found that this was true only on heavily loaded server systems running a number of CPU-intensive processes that is larger than the number of installed physical CPUs. In our workloads, we typically have only as many compute tasks as CPUs.

The Linux kernel views each hyper-threaded processor as two virtual processors. When a task is runnable, it is assigned to one of the virtual processors. Ideally, since we have two physical processors per node, the kernel should schedule each task to a different physical processor. Unfortunately, we found that in many instances, the kernel scheduled both CPU intensive tasks to virtual processors on the same physical processor. The resulting cache misses and contention significantly reduced our systems performance.

In light of this finding, we have turned off Hyper-threading on most of our nodes.

Stability

IBM describes the HS20 cluster as: "Powerful 2-way Intel processor-based blade server delivers uncompromising performance for your mission-critical applications." [6] We purchased this equipment in 2003/2004 from IBM because we valued IBM's reputation for reliability and service. We have since had cause to regret this decision.

Each of our blades came mounted with two Toshiba MK4019GAXB [20] 40 GB disk drives. These drives are 2.5" hard disk drives of the type

found in many laptops. IBM probably choose to use 2.5" drives because of the size restrictions imposed when fitting two drives on each blade.

In the interest of reducing disk management, each PlanetLab node mounts its own local disks as a single large logical volume. This construct enables the system to allocate disk space without concern for the size of each partition or disk drive. The drawback is that if any of the drives fail, the whole partition is corrupted and lost. Recall that in PlanetLab, it is easy to re-install a node. The loss of any one node is pretty trivial and the resources can be easily recovered (although the data on that node is lost).

At installation time, the Evergrow clusters had a total of 244 MK4019GAXB drives. In the first three years of ownership, we have replaced more than 50 of these drives. By far, the most common problem and the largest management headache has been the failure of disk drives. We have not identified any common cause behind these failures. Some of our clusters are in very professional data centers with significant cooling capacity and managed power. Other clusters are in less professional locations with limited cooling and whatever power is available from the wall socket. Location does not seem to be a factor in the failures. We can only conjecture that these drives were defective in one way or another. There have been no failures with any of the non-MK4019 replacement drives.

Monitoring

Monitoring of resources is an important part of any research project. At a minimal level, monitoring tools identify software, systems and hardware that may not be operating as expected. One level up, monitoring provides a list of assets that administrators and users can reference to identify and find available resources. At the management level, the project coordinators can watch the monitoring systems to identify users that are utilizing the projects resources. This data also delineates those registered users who are not using the system.

Ganglia

We began to wonder about utilization of our new system about six months after our partners received their cluster hardware. We knew that each partner had deployed the systems, but we had no idea if the clusters were in use or even if they were operational. Our very first effort was to implement a centralized monitoring system using the Ganglia [10] software package. Ganglia provided at-a-glance status for each cluster and each node.

Ganglia implements both push and pull communications. Each monitored node runs a ganglia daemon called gmond which intermittently collects information about the local system state. Each gmond daemon sends a regular update to a gmeta collector daemon. The gmetad daemons maintain a list of all clients that

sent it data along with the details for those clients. We have one gmetad daemon for each cluster and a separate one for the EverLab system. In our implementation, a central gmetad daemon running on our main server polls each of the gmetad daemons on a regular basis and collects a snapshot of that daemons stored status. The collected data is then stored in RRD [13] databases for graphing and presentation.

We keep a web browser open to our local Ganglia monitor. At a glance, we can see which clusters are operational, busy, or down. Ganglia provides summary statistics of Total, Up and Down hosts which quickly reflect the overall system health.

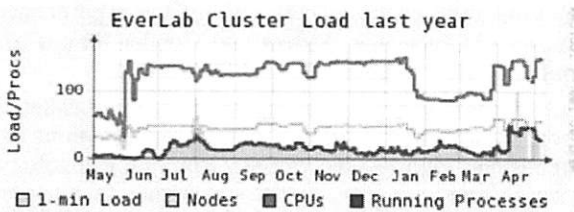


Figure 5: May 2006 to May 2007 EverLab One Minute Load Average.

The main Ganglia load graphs display the 1 minute load, the number of nodes, the number of CPUs and the number of running processes. An optimally utilized system would have one process for each CPU. The Everlab One Minute Load Average graph shows the 1 minute load on the EverLab system between May 2006 and May 2007. The line at the top of the graph shows the total number of CPUs, some of which are actually virtual hyperthreaded CPUs. The second line shows the number of reported nodes and the third line shows the number of running processes.

As can be seen, the number of nodes changes over time. The majority of these outages are related to power and cooling problems in the remote data centers. The graph also shows the growth in EverLab usage. Initial use was minimal until after the EverLab Workshop in June of 2006. From that point until March 2007, we averaged about one process for every two nodes. Toward the end of this period, we saw usage increase to approximately one process per node.

Ganglia has been very useful in Evergrow and EverLab, but was not particularly successful in the PlanetLab environment. In 2001, one of the Ganglia developers joined the PlanetLab project and began deploying Ganglia over PlanetLab. Over time, the Ganglia installation was removed and forgotten. It seems that Ganglia did not provide enough benefit to the PlanetLab team to warrant its maintenance costs.

We believe Ganglia was appropriate for EverLab but inappropriate for PlanetLab for of the following reasons:

1. EverLab is naturally organized by clusters of nodes. The gmond daemons communicate over

the local network to their gmetad parents. In PlanetLab, there is no natural structure and hence each gmond must send remote messages to a centralized gmetad. This increases the rate of lost message and requires significant bandwidth at the central node.

2. PlanetLab nodes have traditionally been heavily utilized. It is very rare to find a PlanetLab node that has no running processes. Ganglia shows that their nodes are busy. Ganglia provides dozens of detailed metrics, but does not differentiate between slices. From a high level perspective, We use Ganglia only to show that a node is busy or down. The detailed metrics have not been useful in our environment.
3. PlanetLab has more than 600 nodes. Ganglia does not scale particularly well over a few hundred nodes. For example, in Evergrow, the Ganglia web front end periodically queries the main gmetad daemon for the system state. The data is returned as a single large XML document at least 220K bytes in length. The equivalent file for PlanetLab would be more than 1 MB large. Sending this data over the wire every 60 seconds is inefficient, particularly since most of the data has not changed.

We are very pleased with Ganglia for our size installation and would be likely to choose it again. For us, the benefits of a simple monitoring system outweigh the network traffic overhead. Ganglia is a rough tool. The instantaneous data is frequently incomplete because of the way that the gmetad daemon collects and updates its internal data structures. Improvements in this area would make Ganglia more useful and reliable.

EverStats

Ganglia provided our system with cluster and host level monitoring. We could identify node status and utilization. But our project administrators wanted more. They asked us about the users and their projects. Which projects were deployed on EverLab? Were the projects computationally focused or more experimental? How many unique projects were actually using the system? To address this issue, we developed the EverStats usage monitoring system.

Our first challenge was to collect data from the EverLab nodes. We knew that the CoMon [15] project had developed a tool on PlanetLab to monitor node usage, so we borrowed their underlying monitoring tool called slicestat [14]. The slicestat daemon runs on each PlanetLab node and like the gmond daemon collects performance metrics. Slicestat has the added benefit that it understands the PlanetLab virtual server architecture and can report data according to each slices activity.

The CoMon project polls each slicestat and provides current CPU data along with 1 minute and 15

minute network statistics. In many ways CoMon overlaps with Ganglia as a monitoring tool.

Our interest was not in the short term node and slice status, but in the historical usage patterns. We installed slicestat on all EverLab nodes and then built a custom tool called EverStats to poll the daemons and store the results in a long term database. EverStats provides summary reports on usage for the past week, month and year and allows administrators to drill down from both a slice or node view.

In order to minimize network traffic, EverStats polls the slicestat daemons once every five minutes. We keep the raw data for 24 hours and then summarize it as daily data in our database. Short term activities such as running a program for 30 seconds are unlikely to be visible on Everstats. On the other hand, computation or experiments that run for a hour or more will certainly be reported.

There are more than 65 unique projects registered in the EverLab database. New users tend to create a test slice to familiarize themselves with the system. These users then move to a project slice that is shared by their research team.

The EverLab slice groups report reports on the cumulative data for all slices in each defined group. The sample EverStats Slice Group Report shows a representative report from April 2007. The System group represents the basic EverLab services. These services tend to be active for short periods of time, but are visible when a node is idle.

The Condor slice group represents the distributed Condor instantiations on each of our nodes. Condor is currently under very light load and hence the values represent a sort of steady state overhead similar to the System group.

The rest of the groups represent research originating at their respective universities. The "Others" group is a catch-all for research from one of our non-cluster partners. There are currently seven slices in the "Others" group. The university research groups have between two and five slices each.

As can be seen from the report, most research projects do not use the full power of EverLab. As the number of nodes in an experiment increases, so too

does the complexity of deployment, debugging and monitoring. Projects tend to use the maximum number of nodes necessary to produce a reasonable academic paper.

Of the six project groups, more slices are CPU bound with CPU loads running between 88.44% and 261%. A factor over one indicates that there is more than one process running concurrently in these slices. Experienced High Performance Computing researchers attempt to allocate exactly one process to each processor in order to decrease contention for CPU cycles

The "Others" group provides a good example of slices involved in networking experiments. These slices are generating on average 45Kbps of incoming and outgoing traffic over the life of the experiments. As can be seen, the System and Condor groups are minimal users of both CPU and Networking.

EverStats is a useful tool in its current instantiation. Potential extensions include the graphing of trends and improvements in the sampling technology. Graphs and Trend analysis would be helpful for presentations and for tracking the natural growth and decline of project activities.

The current sampling technology serially queries each of the nodes. In addition to being inefficient, this approach takes a significant fraction of the five minute query period. While acceptable for EverLab, the serial query mechanism takes more than thirty minutes on the full PlanetLab for each query cycle.

Ideally, we would like to see EverStats integrated into the base CoDeen and CoMon projects for use by both Private and Public PlanetLabs.

Education

The Evergrow project is composed of researchers in Computer Science and Computational Physics. All of our researchers are computer literate and have some scientific programming ability. At the beginning of the Evergrow project, we assumed that researchers would use any and all computational resources that we could provide. In fact, we found that computational resources are currently widely available. Desktop workstations have enough processing power to handle many tasks previously allocated to dedicated processors. One of our

Slice Name	Nodes	Total CPU Hours (all nodes)	Average % CPU	Avg. Outgoing Bandwidth (Kbps)	Avg. Incoming Bandwidth (Kbps)
System	52	2129.15	0.72	0.04	0.06
Condor	52	1086.05	0.96	0.21	3.40
Aston	34	19048.40	97.23	0.00	0.00
HUJI	25	5738.09	96.75	0.01	0.67
Orsay	10	2729.79	261.93	0.00	0.00
SICS	19	1575.53	88.44	2.82	1.14
UCL	25	4071.05	134.48	18.18	1.61
Others	52	5088.41	8.32	44.70	46.69

Table 1: Sample EverStats slice group report.

authors processes gigabytes files on his laptop. The performance is not great, but the benefits of taking your work with you outweigh the time to completion.

Our partners provided a list of requirements when we started the Evergrow project. Some wanted High Performance Computation (HPC) services. Others wanted distributed network platforms for experimentation. The resulting EverLab system provides both, but we found that our partners needed help getting up to speed. Our most effective tool has been hands-on workshops. Our first workshop was held in June, 2006.

PlanetLab (and EverLab) present the world as set of virtual servers running on remote hosts. At one time, using telnet, SSH and X-windows was the standard method for interacting with remote hosts. Today, undergraduate and graduate students use the Microsoft Windows platform and Microsoft Remote Desktop connection. PlanetLab's interfaces are much more basic and are less familiar to many of our partners.

During our workshop, we walked the participants through the EverLab process. To get started running your own code on EverLab (or PlanetLab), a researcher must:

1. Have a registered site and Principal Investigator (PI). We created a pseudo-site called EverLab for all of our users.
2. Request an account by filling out a web form.
3. Wait for the account to be enabled by the PI or site administrator.
4. Create and upload an SSH key to the management web site.
5. Have the PI create a slice for your project.
6. Assign the users to the new slice.
7. Assign the slice to one or more nodes.
8. Wait until the slice propagates to the target nodes.
9. Log into the slice using SSH.

The total latency from start to finish is minimally about one and a half hours. For a user trying this remotely, it can take between one to three days just to be able to log into the nodes. The major benefit from our workshop was to shorten this initial period and to get users working during workshops' first day. The second day was spent learning about Ganglia, Condor and custom deployment scripts that other researchers have written for deploying experiments on PlanetLab/EverLab. Summary of presentations and tutorials are available on the web [2].

We have found that all of our active researchers attended our workshop. It may be that other researchers do not need our dedicated resources, or that the learning curve is too steep. We plan on continuing our educational efforts and working with our researchers to identify the barriers to better system utilization.

Future Work

We have identified a number of areas for future work on PlanetLab in general and in particular on

EverLab. Each of these areas are extensions of our experience with the current EverLab system and its user community.

Security

Fedora Core 2 (FC2) was first released in May 2004. Fedora Core 4 was the more recent release as of September 2005, when we started working on PlanetLab. As of the summer of 2006, EverLab is still running FC2 on its nodes and FC4 on its central management node (PLC).

Fedora Core 2 officially reached its end-of-life in June 2007. Fedora Core 4 reached its end-of-life in January 2007. Fedora Core 5 was retired in July of 2007. The implication is that bug releases and security patches for these systems are no longer available from the Fedora team for these systems.

Our experience and that of the PlanetLab Consortium is that there have been almost no security issues related to FC2 or FC4. The latest PlanetLab V4.0 release still supports only FC4. Common wisdom would suggest that we update to FC7 as soon as possible. Our experience has been that our deployed version of the three year old FC2 has been stable and secure and that there is little urgency to upgrade.

Usability

Our user community differs from the standard PlanetLab community in their grasp of UNIX tools. The PlanetLab community includes many systems researchers who understand the Linux operating system and its user level tools in detail. Our community of physicists and computer science theoreticians do not have this level of systems knowledge. We would like to see future systems include a suite of basic, easy to use tools for accessing the nodes, deploying applications, collecting logs and monitoring the experiments activity. In most cases, the effort to develop these tools is one of packaging and documentation.

Improved Coordination

With the release of PlanetLab V4, there has been significant improvement in the installation and upgrade processes for private PlanetLabs. The major remaining issue is coordination on PlanetLab changes. We would like to see a PlanetLab Engineering Task Force (PETF) that would manage platform changes and coordinate platform security.

We see PlanetLab as a moving target. There are many possible ways to extend and improve the system. The challenge is to choose the appropriate changes for the private PlanetLab community. Private PlanetLabs value stability and security over experimental features. The PETF would collect and document these changes. It would provide a repository for all blessed changes and versions of the system.

As a production system, PlanetLab should have a security coordinator. The PETF would track published and zero-day attacks on PlanetLab or its constituent

components. It would provide timely notification and updates to administrators concerning these attacks and would coordinate efforts to detect and correct these problems as they occur.

The PETF could organize workshops and conferences for Private PlanetLab administrators and users as a way to educate the community and to identify areas for improvement and growth.

Federation of PlanetLab's

One vision presented by the PlanetLab community is to integrate remote PlanetLabs in a federation. Users on one system would be able to utilize resources on federated systems while abiding by inter-system usage policies. This concept requires coordination of the detailed federation interfaces as well as the definition of appropriate system level policies.

The EverLab installation has many of the features required for a federated PlanetLab system. We operate in a production environment and our system is not over-subscribed. While our project would welcome federation with other private PlanetLabs, our partners network administrators would be hard-pressed to open the system to non-partner sites without additional controls to protect their networks from abuse.

Conclusion

Everlab serves as a model for future research efforts. It bridges the gap between Grid based HPC installations and free-for-all experimentation systems. EverLab makes efficient use of administrative resources and provides reporting services to monitor system usage, reliability and responsiveness. EverLab provides a service for setting policy on resources so that all participants have access to the shared resource. With the exception of pure HPC projects, We believe that future efforts should include an EverLab style system for system management, resource allocation and monitoring.

Acknowledgement

We would like to thank the PlanetLab Consortium based at Princeton for its extensive support and encouragement, without it the EverLab project could not have been accomplished. Special thanks to Consortium members Marc E. Fiuczynski and Steve Muir. Lior Ebel was instrumental in developing EverStats.

Author Biographies

Elliot Jaffe received his B.Sc. in Mathematics from Carnegie Mellon in 1985. He worked in industry as a system administrator, developer, integration specialist, manager and CTO. He returned to academia and received his M.Sc. in Computer Science from The Hebrew University in 2005 where Elliot is currently working towards his Ph.D. in Computer Science. Elliot's research interests are in the areas of Software Engineering, Distributed Systems and Large Scale Storage.

Daniel Bickson is currently a Doctoral candidate at the School of Engineering and Computer Science at the Hebrew University, Jerusalem, Israel. His research interests include Communications, Network Security, Distributed Systems and Belief Propagation.

Scott Kirkpatrick has been a Professor in the School of Engineering and Computer Science at the Hebrew University, Jerusalem, Israel since 2000. While at IBM Research in his previous career, he compiled a distinguished scientific record (90+ publications, 10+ patents) and was elected a Fellow of the AAAS, the APS and the IEEE. Some of his papers are among the top cited of all time, in both Physics and Computer Science. In addition, Prof. Kirkpatrick has more than 20 years of experience in management at IBM, supervising several large multi-team research and development projects, and currently coordinates the 25+ Partner FP6 IP EVERGROW (2004-2008).

Bibliography

- [1] Bickson, D. and D. Malkhi, "The Julia Content Distribution Network," *The 2nd USENIX Real World Distributed Systems (WORLDS '05)*, 2005.
- [2] Everlab Workshop, Huji, Jerusalem, Israel, June 7-8, 2006, <http://www.cs.huji.ac.il/labs/danss/p2p/evergrow-workshop>.
- [3] Foster, I., C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal High Performance Computing Applications*, Vol. 15, Num. 3, pp. 200-222, 2001.
- [4] Foster-Johnson, E. (Red Hat), *Red Hat RPM Guide*, March, 2003.
- [5] Gropp, W., "Mpich2: A New Start For MPI Implementations," D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *PVM/MPI*, Vol. 2474, *Lecture Notes in Computer Science*, p. 7, Springer, 2002.
- [6] IBM, *IBM Blade Servers – BladeCenter T-HS20 Server*, http://www-03.ibm.com/servers/uk/eserver/bladecenter/hs20/more_info.html.
- [7] Intel, *Preboot Execution Environment (PXE) Specification*, 2002.
- [8] Ligneris, B. D., "Virtualization of Linux-Based Computers: The Linux-vserver Project," *19th International Symposium on High Performance Computing Systems and Applications, HPCS 2005*, pp. 340-346, May, 2005.
- [9] Lukic, J., E. Marinari, O. C. Martin, and S. Sabatini, "Temperature Chaos in Two-Dimensional Ising Spin Glasses With Binary Couplings: A Further Case For Universality," *Journal of Statistical Mechanics*, 2006.
- [10] Massie, M. L., B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation and Experience," *Parallel Computing*, Vol. 30, Num. 7, July, 2004.

- [11] Morato, D., E. Magana, M. Izal, J. Aracil, F. Naranjo, F. Astiz, U. Alonso, I. Csabai, P. Haga, G. Simon, J. Steger, and G. Vattay, "The European Traffic Observatory Measurement Infrastructure (ETOMIC): A Testbed For Universal Active And Passive Measurements," *TRIDENT-COM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMMunities (TRIDENTCOM'05)*, pp. 283-289, IEEE Computer Society, Washington, DC, USA, 2005.
- [12] Napier, D., "IPTables/NetFilter – Linux's Next-Generation Stateful Packet Filter," *SysAdmin: The Journal for UNIX Systems Administrators*, Vol. 10, Num. 12, pp. 8-16, Dec., 2001.
- [13] Oetiker, T., <http://ee-staff.ethz.ch/~oetiker/webtools/rddtool/>.
- [14] Park, K. and V. Pai, <http://codeen.cs.princeton.edu/slicestat/>.
- [15] Park, K. and V. S. Pai, "Comon: A Mostly-Scalable Monitoring System For Planetlab," *SIGOPS Operating Systems Review*, Vol. 40, Num. 1, pp. 65-74, 2006.
- [16] Second European Planetlab Workshop, EPFL, Laussane, Switzerland, October 27-28, 2005, <http://lsirwww.epfl.ch/planetlabeverywhere/>.
- [17] Shavitt, Y. and E. Shir, "Dimes: Let the Internet Measure Itself," *Computer Communication Review*, Vol. 35, Num. 5, pp. 71-74, 2005.
- [18] Spring, N., L. Peterson, A. Bavier, and V. Pai, "Using Planetlab For Network Research: Myths, Realities, and Best Practices," *SIGOPS Operating Systems Review*, Vol. 40, Num. 1, pp. 17-24, 2006.
- [19] Thain, D., T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience: Research Articles," *Concurrent Computing: Practice and Experience*, Vol. 17, Num. 2-4, pp. 323-356, 2005.
- [20] Toshiba, *Mk4019gax*, <http://www.sdd.toshiba.com/main.aspx?Path=818200000070000000010000659800001516/818200000aff000000010000659c000026ad/818200000192000000010000659c0000279f/81820000019f000000010000659c0000054e>.
- [21] Ripeanu, Matei, Mic Bowman, Jeffrey S. Chase, Ian Foster, and Milan Milenkovic, "Globus and PlanetLab Resource Management Solutions Compared," *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, Honolulu, Hawaii, June, 2004.
- [22] Intel, *Hyper-Threading Technology*, <http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>.

Master Education Programmes in Network and System Administration

*Mark Burgess – Oslo University College
Karst Koymans – Universiteit van Amsterdam*

ABSTRACT

We report on and discuss our experiences with teaching Network and System Administration at the level of Masters at Oslo University College and the University of Amsterdam. At our respective institutions we have independently arrived at very similar models for teaching a traditionally vocational subject within an academic Computer Science framework by incorporating a strong practical component.

Introduction

For several years now academically inclined system administrators have struggled to identify the role and place of System Administration within the fields of Computer Science and Engineering. This effort has often brought controversy, with strong and diverging opinions dominating over any consensus. This is not uncommon for a novel field of study. Whereas many university subjects evolve from academic and vocational traditions that have existed for generations, the establishment of a curriculum in System Administration (a hybrid subject somewhere between computing, science and engineering) cannot make the progress industry and society needs by waiting for the results of a protracted evolutionary process. The need for system administrators is here now.

In our degree programmes we have chosen to avoid the controversies and formulate courses based on our own experiences as system administrators and academics. We believe that professional educators have a better chance of settling these controversies than those whose passions rage, as we shall explain below. By anchoring the subject in established academic traditions, but maintaining the hands-on aspect of the subject, university milieux have been more receptive to the idea of system administration as an academic discipline, even when they have not always understood the initial vision. What is interesting is that, in spite of the lack of standardization in thinking, and in spite of superficial practical differences imposed by our local environments, the courses we have developed in Oslo and Amsterdam overlap strongly in both flavour and substance, so perhaps consensus is not so far from reality after all.

Our aim in this paper is to report on our efforts in this area. We do not present our courses as perfectly formed, finished products to be admired by all (no university course ever turns out the way its designers would like, due to numerous constraints and obstacles), rather we comment on the philosophies and implementations that have led us to make courses at our

Universities. In each case, although we each have some relevant introductory Bachelor level courses, we have found that Masters level study programmes are most suitable for implementing system administration studies, since students benefit from Bachelor level skills in more standard computer science as well as from a certain breadth of background. We shall not attempt to view system administration as a profession in the sense of apparently similar organized job descriptions (like the medical profession), as this topic is charged with issues that go far beyond education. Rather we focus on our experiences as educators and point out how we have approached a problem that some have claimed was impossible: to turn system administration into a discipline.

We begin by discussing our approaches to educating system administrators, then we consider norms and standards in related fields. We summarize the content that we consider to be essential and report on our experiences with this. Finally we draw some initial conclusions about our successes and failures. We hope that the strong similarities we have arrived at in our own neutral attempts to formulate system administration academically will help others to see their own subject or profession through more impartial eyes, and perhaps even help to advance the state of understanding of the field.

Relationship of System Administration to Computer Science

Let us begin by asking how system administration relates to other fields of research that are commonly associated with it. Computer Science derives traditionally from two camps: mathematical logic and electrical engineering. Both of these have long academic traditions that have influenced the way computing has been researched, framed and is taught in Universities. In addition it is known that physics graduates are well represented in some areas of Computer Science. Not everyone in working with computers has learned the subject in a university. Computing as a

phenomenon is young and many self-taught hobbyists have entered the workplace on the strength of their own private initiative. Such people ought not feel threatened or belittled by academic initiatives.

System administration has not normally been a subject in its own right, though many short courses have been offered by universities around the world. However it has not been completely absent from curricula. Some work that has gone into studying "computer management" from universities and graduates has been in the area of telecommunications, thanks to often generous sponsorship of powerful telecom organizations. Much of this work has been closed-source however and centered around organizations like the Telemanagement Forum (TMF) and the Internet Engineering Taskforce (IETF). It goes back to the 1970s under the title of "Network Management" and, in some respects, many of the issues facing system administration have been discussed and "solved" within that limited context, e.g., see [1]. Thus, system administration lags behind its related "big brother" Network Management. Many electrical engineers who have found themselves in the clutches of information technology revolution have entered this field through telecommunications. It is taught in various courses especially in Europe (where most of the research in this field is carried out) and it is represented by major conferences like NOMS [2] and IM [3] and the IARIA [4] conferences, organized by major telecommunications and routing companies like Cisco and Motorola. Since about 2001 we have made a concerted effort to cross-pollinate these disparate communities.

A second group that has involved itself in management concerns is software engineers. Distributed software systems and middleware are often used to "manage" software layers, instrumenting software with inter-communication capabilities that need to be managed just like more complete operational environments. Although the list of challenges is somewhat smaller in software engineering, and this gives software engineers an oversimplified impression of the challenges of system administration, the overlap makes a connection with software engineering. It also contributes to the widespread belief amongst computer scientists that system administration can be solved through software engineering alone, though this seems to be changing as computer systems become more ubiquitous (see conferences like DSOM [5] for these communities).

How Should System Administration Be Taught?

The need for formal education is rarely disputed but the form is often controversial, especially amongst those who learned some of the necessary skills flying by the seat of their pants. It is common for self-taught practitioners to reduce subjects to a list of a few skills that are needed, then "the rest is experience." It has even been suggested that it is "too early" to formalize anything about the field, since it is not yet well enough

understood. However, the job of a skilled educator is to pragmatically distill such experience into a literature of material that can be taught, and we believe that it is never too soon to do this. Sufficient understanding is a journey rather than a finished product.

There is no single approach to learning that can solve all of a society's needs. The need for learning does not disappear once we are finished with school and so there is a need to combine further education with work in some appropriate way. As we shall discuss below, society is becoming less tolerant of paying for schooling, and is increasingly pressuring both enrolled students and would-be students to work alongside their further education.

Education is typically covered by a three-pronged strategy which involves:

- Self Learning.
- Training.
- College/university studies.

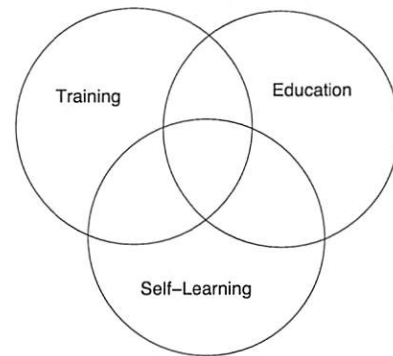


Figure 1: The three aspects of learning.

Although most education in system administration is in the first two categories, it was our intention to formalize the subject into an academic discipline in our master degree programmes. This has been a considerable challenge and has taken the better part of ten years of preparation in both our groups.

Our colleges have contributed several important text books in this field, one at Bachelor/Master level [6] and one at Masters/Ph.D. level [7]. A book for bachelor level studies also covers the Norwegian market [8].

Self-Learning

Self-learning is learning without the guidance of a tutor or course coordinator. There is no assessment and no interactive diagnosis of a student's progress. Self-learning is clearly a necessary part of any learning scheme learning without some effort by oneself would require technology yet to be invented (some would say that teaching is, in fact only management, that all we do is promote an environment which accelerates the student's own learning process by providing for, inspiring them and guiding them). Self-learning however implies that a student does not have access to

an organized programme of study and is therefore lacking in the potential benefits of others' experience.

Self-learning is usually the only option available to the would-be student at the outset of a new field of study. In engineering disciplines particularly it remains an important aspect of a learning strategy, in the guise of "trial and error" practice. Getting one's hands dirty in the field is one of the most important confidence building exercises, and it is a fast track to connecting experience to meaning. Trial and error is an efficient approach to knowledge acquisition because one can easily see the patterns of success and failure at first hand. Both authors have had their share of self-learning experience.

Self-learning is not the same as experimentation or lab work. Many technicians working in laboratories are simply carrying out practices procedures that are often later replaced by automated machines. There does not need to be understanding to complete a task. Self-learning implies a development of understanding through one's own effort. The effort might simply involve reading, but in most cases it requires a person to engage their motor functions and do something physical (whether the doing is simply writing, solving math problems, or plugging cables into boxes).

We might define a subset of self-learning to be self-training, which we understand as the study of recipe-solutions to problems from standard materials. This is a common form of training for certification programmes. Self-learning can therefore span both education and training, in the following sense: significant reading can lead to a broad perspective on problems if the reading is broad enough and the self-taught student has the opportunity to put the reading into context.

Training

Training is a form of targeted knowledge presentation often aimed at teaching skills or giving summary overviews. It can never be a substitute for a complete education; it is mainly useful for filling in gaps in a basic knowledge base, such as learning a particular procedure or using a new tool. Training is not a realistic strategy for teaching complete and 'rounded' professionals because there is inevitably a need for long-term integration of knowledge into one's existing cultural base.

For people in full time employment, training is often the only available alternative to interact with a teacher. Short courses given at conferences or by other providers offer an access route to organized classes when students have time limitations, such as during full-time employment. Training courses are usually based on a set of slides and are presented at a low level for a wide audience, since they are financed by their commercial success. This leads to a side-effect of training which is an unintentional "dumbing-down" of the material in order to reach the largest paying denominator.

Some companies offer training programmes to graduates because they see gaps in their knowledge in specific skill areas. A graduate already has a basis of deep knowledge into which a training course can be integrated. A notable training programme that exceeds most is the Cisco Academy/university training programme.

The traditional form of training is that of half-day to one-day classes at conferences such as LISA, USENIX, NOMS [2], IM [3], etc. These are slide shows with commentary. Training tends to be supplied as a list of recipes and do's and don'ts. There is little time to develop any significant understanding. If an idea does not resonate with the trainee more or less immediately, he or she has little recourse other than recommended texts to fill in the gaps.

Training can be a "leg up" to help a motivated employee to self-learn for instance. This motivational aspect of teaching should not be underestimated. Training at conferences forms part of a (hopefully) positive experience that can have a powerful motivational effect. It is also sometimes accompanied by certification to capture part of the benefits of verification. Certification involves some kind of test, normally a multiple choice psychometric evaluation. Verification and testing can be employed in principle to ensure that trainees reach a standard interpretation of the subject of training, rather than their own (perhaps distorted) version.

College Studies

Accredited college or university education is the oldest kind of higher education, the most extensive and the only kind of learning that addresses all of the strategies for learning at the same time. A planned educational programme in a college or university has the potential to incorporate elements of both of the foregoing approaches within an extended curriculum. The ability to test students repeatedly serves also the purpose of measuring their abilities and motivating them to improve their performance.

A college education can build up concepts and principles over an extended period of time, and place them in a larger context. This is an important cultural experience. Humans learn essentially by story-telling, in which they must form their own version of a story, and place themselves within it, before they are willing to believe in it fully. An extended programme can explore with students the reasons behind what they are learning and build their own personal motivation. In a college programme, students have time for trial and error and they have the opportunity to put knowledge into a context.

College education is more interactive than either training or self-learning. There is a greater opportunity for feedback, building confidence and quickly correcting misconceptions. Moreover, the experience has an epidemic effect – one conversation with a student can

improve the teacher's understanding and spread to another students, and so on in a viral way.

Another phenomenon introduced by college education is the concept of "group self-learning," where students study in the alternating role of teacher and trainee. This so-called "power learning" is a useful extension to the materials and wisdom brought in by the official teacher. Both our institutions have practiced this with success.

Pre-requisites and the Culture of Learning

So what ought students know before and after their education? As university lecturers, we affirm that the issue of what students ought to know in advance of a course programme is highly politically charged. Education is a cultural phenomenon and priorities are constantly changing (not always for the better). Colleges and universities inherit students from other schools and workplaces, and cannot guarantee that applicants will reach the minimum bar expected for starting.

One difference between our study programmes lies in the attention to pre-requisites. At Amsterdam, far more work is put into selecting strong applicants and deterring weaker candidates than at Oslo, because the curriculum is taught in half the time and under greater pressure. An additional year for personal growth allows the course at Oslo to pull through students who might not make it in the pressure-cooker model used at Amsterdam.

Both institutions require basic programming, knowledge of operating systems principles and some Bachelor level mathematics. No institution can ever teach everything someone might need to know about a subject and thus colleges and universities do not try to do so. They do not usually give "training" in specific skills except as an example to a more general discussion. Rather, the approach is to charge students with the more fundamental skills needed to learn for themselves, along with a critical eye so as to not take everything at face value. Educational institutions have developed strategies for teaching these general skills over centuries. These methods have become cultural norms, and there are many reasons why we should respect them as norms, even if they do not train students to specifically use tool X or carry out procedure Y.

Teaching educational norms is important especially because it results in graduates who can communicate in a cross-disciplinary field and society at large. Examples of teaching norms include basic science and language skills (reading, comprehension, reporting, creative writing, summary etc.). Mathematics and physics are taught, for example, not usually to allow people to calculate planetary orbits, but because the skills one must surmount to complete these topics have general validity. Moreover, since most other people have been through the same experience, it becomes shared knowledge and this allows any student to communicate with

any other person who has learned the same set of standard concepts. ("Remember how we used to calculate the efficiency in physics? We could do the same thing here...") This would not be possible if the training were too specific and too directed. The ability to write clearly and to reason about problems is something else students of mathematics and physics learn. Writing and language skills have many of the same features of mathematics: grammatical structure, attention to detail, the need to interpret the meaning from a potentially ambiguous signal, and so on.

This cultural aspect of education is under pressure today from impatient and under-educated spokespersons in our societies who would dumb it down. The urgent rush to specialize and "train" people for "the jobs industry needs today" eventually becomes harmful to the culture of learning in society. If we bypass well-known metaphors in favour of new short-cuts, sometimes using computer software to remove the need even for a basic skill to be learned, then we impoverish the experience. We need these cultural norms in education, like math and science, both because they allow us to speak a *lingua franca* of reason that crosses discipline boundaries, and also because they allow us to re-use the experiences and hard-won understandings of other fields.

We believe it is important to understand the limitations of basic training: a trainee can repeatedly complete a task without any understanding whatsoever. It is only when the train runs off its tracks somewhere that one discovers that the uneducated student has no idea where he or she is in the landscape of knowledge.

And this is exactly where the need for academic level education comes in. Not only to lay out the tracks and keep the train on those tracks (which can be trained more or less in the above mentioned education shortcut), but also in the case of derailment or when entering unknown territory.

Modeling

The philosophy of science gives us one of the most essential tools in the problem solver's toolkit: the idea of modeling. As the philosopher of science David Hume maintained, there are two kinds of knowledge that should be clearly distinguished.

- Theoretical knowledge that can be determined precisely and exactly (proven perhaps), but whose relationship to the real world is uncertain.
- Empirical knowledge that is certainly about the real world, but whose measurement and interpretation are not exactly determined.

In both cases our understanding of the world is imperfect. Our basic understanding of phenomena involves building approximate mental models, so a deeper understanding of this process can only help students towards a deeper understanding of the phenomena also.

A model is composed of suitably idealized approximations that attempt to manage this uncertainty and allow one to:

- Make predictions.
- Calculate answers.
- Classify and hence understand phenomena.

These are important parts of rational thought.

However, we are also faced with problems in this process. The usefulness of standardizing knowledge, so as to make it more authoritative, comes at the cost of whitewashing over these uncertainties. Science itself has to some extent become standardized and even commercialized in curricula today, so that it is often taught misleadingly. Students come to attribute science an almost mystical reverence as they like many in society harbour the erroneous belief that science teaches “truth” rather than approximate models of practical value.

There is a lesson here: shrink-wrapped packaging of education which violates the questioning and critical spirit of scientific discovery can breed ignorance as quickly as it can teach skills. These are important skills in system administration. It is vital that students be shown how to ask fundamental questions and be able to question assumed truths. Nevertheless there is a role for standardization in the language used to describe a subject. A common ontology is as important as shared cultural values in enabling students to communicate without talking at cross-purposes.

Standard Curricula for Computer Science

The Ironman Curriculum effort started by the Association of Computing Machinery is an effort to standardize the framework of topics in a consistent taxonomy. This has been a long term effort. The complete Ironman report includes a number of documents, e.g., [9]. These are far too numerous to discuss here. They can all be found at the ACM website.¹ System administration was initially absent from the Ironman documents, which have been growing since the late 1990s and were finalized only in 2005. Several terms have now entered into the curriculum.

It is interesting to see how terms are integrated into a traditional computer science framework. While some system administration activists would apparently prefer to define system administration as an entirely separate enterprise, here we see that topics have been slotted into the existing taxonomy of categories rather than defining it as a tumour to be tacked on to the edges of computer science. This partly reflects the approach to the subject taken in Oslo, where some compromise on words and terms has been made to integrate the knowledge into the larger picture of programming, Unified Modeling Language design, database searches, service orientation etc. However, there are topics in system administration that do not traditionally

appear anywhere else in computer science: fault management, reliability, policy etc. These are also existing disciplines that overlap system administration with other fields, possibly in different university faculties.

On closer examination the course material in our Masters programmes fits (surprisingly) comfortably into the Ironman standard curriculum as long as one reads it with appropriate glasses. This means that even colleges without an explicit degree in system administration could put together a helpful syllabus that would be relevant to the field, with the help of some massaging of language and examples.

From Principles to Content

We turn now to the content of our degree courses. What are the key areas that constitute an education in system administration? Previously the System Administrator's Guild SAGE, with the special assistance of Rob Kolstad, has proposed to build a taxonomy for system administration. Other possibilities for mapping knowledge have been proposed recently with the growth of interest in semantic webs: the concept of ontology is presently quite popular [10]. An ontology goes beyond this with an extended list of terms, usually belonging to a single and uniform cultural body. So far however, this has not formed a useful basis for an educational map where common concepts bind topics together rationally. One reason for this could be that taxonomies are inherently subjective and such subjective impressions and focal points change very quickly in the technological disciplines and create more controversy than consistency.

At Oslo university College, our approach has been to search for a stable core that can be used to teach understandable models and principles, and then to pepper this core with contemporary examples and practice, because understanding is based on models. The course structure was built from the Bachelor level up by identifying common principles from a mass of empirical writing and practice [6]. However, in spite of having courses at Bachelor and Masters level, there are topics that we cannot cover sufficiently for everyone's taste. For example, there is probably sufficient material to give an entire course on “storage” (quite desirable in present times), but this would only mean less time for something more fundamental and the time might simply be wasted when next year the technologies were different.

We find that, in spite of good intentions, we have a particular difficulty giving students a realistic insight into practice. Some basic skills about practice can be learned through laboratory work and research, other skills must be learned implicitly by reading and writing. We are forced to make a judgment about the best use of a student's time for the long run.

An obvious example where an ontology might help to rationalize our approach is in finding a common set of concepts to be used by UNIX administrators and

¹<http://www.acm.org/education/curricula.html>

Windows administrators. Similarly, the chasm of terminology between network management and system administration is so vast that most telecom network management people do not even realise that system administration is a task.

It would be an easy option to give courses in network management, if the idea was simply to fill up a curriculum to attract students. Europe and Asia have dominated the research and commercial activity in Network Management for many years, while the United States has been the greater champion of system administration. Network management has been dominated by the input of software developers using data models to "manage" (usually only to record in a database) information about network devices. The subject is highly protocol oriented and leans towards layered models of centralized management. System administration has been more about UNIX and its highly attractive open environment – giving great freedom to individualists, but consequently lacking focus. An ontological study trying to place knowledge into one cultural framework in such a way that it can be mapped into another, even inexactly, could help to bring these two fields together more quickly.

In putting together courses at our two universities, we have purposely not been led down the paths of least resistance. Rather we have tried to supplement the somewhat bureaucratic views of network management with a more engineering viewpoint. We do not believe that "management" should be equated with monitoring of devices, nor with change management models or database modeling. Instead we have looked for a constructive approach to systems: how to build and maintain them, with a critical eye.

In our chosen course profiles we have effectively proposed our own poor-man's ontologies of most meaningful areas, as many words are inconsistently used, but they are coloured by our own particular tastes and specialties. We shall present some details of these below. As readers will see, the basic ideas chosen by both institutions have emerged along quite similar lines.

Two Year Oslo Programme

In our initial plans for an international two year Masters at Oslo, we expected to be able to require a number of courses in basic system administration related skills, possibly by different names, to the level of our own course System Administration #1. We also hoped also to request some basic computer security. However this wish list soon proved impossible to achieve. With the exception of our own students, there were practically no other student applicants with this kind of background. In security especially, we could see that most colleges offering courses in "security" were really teaching applied encryption, not the kind of rounded security management that we expected.

Consequently we were forced to lower our expectations to admit students with any kind of Bachelor in Computer Science, possessing basic university level maths (discrete math, calculus and matrices) and develop a common framework. We altered our priorities to teach students their missing skills during the master programme. Several approaches have since been tried, including intensive catch-up work in the lab; however, we have ended up by offering the missing courses in their entirety as optional modules, since one cannot digest the concepts in a few lab exercises.

The need for a programme that straddles the dual requirements of an academic degree and a strong practical component is undisputed. The scientific tradition in network and system administration is rather weak, and it has been one of the goals of the Oslo research group to strengthen this. As one of the few institutions carrying out scientific research in this field, we are in a unique position to be able to feed the results of current research into teaching. This happens continuously as new developments and technologies emerge.

One possible approach to curriculum development is to take a mercenary approach and give people what they want. When asked what skills graduates should have, employers are quite unclear however. Some employers would like graduates to be ready-trained to begin work installing a Storage Area Network. Others want graduates who can "think for themselves" and see the "big picture." It was left to the college to decide the curriculum.

We recognized that we cannot teach our graduates all of the skills required to be a successful system administrator. What we hope to achieve is for them to think clearly, constructively and critically about problems, to learn for themselves and be independent thinkers. We have consciously avoided naming courses with specific technologies (e.g., LDAP or APACHE) or skills, except perhaps in the case of the Supercomputing course, which is run by a third party.

Skill-training is probably the weakest part of the courses in Oslo. Students are expected to gain practical skills as a by-product of their work in the laboratory. Some but not all students achieve this. The result is that good students learn both practical and analytical skills, while other students tend to excel in only one or the other. On the other hand, we receive clear feedback from students that the subjects that "change their lives" the most are the more scientifically oriented courses, such as the basic laboratory training and Analytical Methods courses, as many of these basic scientific ideas have never been explained to them before as something they could use.

Course degree programmes are naturally hostage to the general trends in world education. These vary from country to country and we see all variations in our international programme. We must cater to quite different attitudes to learning and skill sets.

Goals

A brief summary of course goals at Oslo follows:

1. Graduates should have an insight into the most important technological developments and scientific results in the field.
2. Graduates should have the ability to apply their knowledge and insight in this field.
3. Graduates should be able to think in the abstract about systems, using the idea of models, generalizations and approximations. They should be familiar with traditional terms, philosophies and concepts of modern scientific thinking.
4. Graduates should know how to search the existing literature of the subject.
5. Graduates should have the capacity to communicate clearly both orally and in writing. They should be skilled in giving clear and comprehensible presentations and be capable of explaining their ideas at the appropriate technical level.
6. Graduates should be aware of societal and ethical aspects of the use of computer technology and its management. They should be able to make ethical judgments and argue for these. They should recognize the difference between an opinion argument and a rational scientific judgment.

These goals can be achieved in a number of ways. We divide the qualities amongst broader subject areas.

Research Based Teaching

Our focus on a strong sense of scientific values has been driven by the desire for our research into computer systems to be of the highest scientific standard (a standard in which Oslo university College leads). Our participation in a European Network of Excellence (EMANICS) also feeds directly into our programme and has motivated several modernisations of terminology and minor changes of focus in the topic we cover. The fact that we focus on general principles, illuminated by examples, allows us to rapidly incorporate changes in current technologies and ways of thinking without completely changing the curriculum.

Our course in High Volume Services, on the other hand, was an area in which we responded directly to a need from industry to system administrators with competency in large scale data center deployment. Such a course did not exist anywhere in the world to our knowledge and thus we instigated a number of research projects which formed the basis of this course [11, 12, 13]. See Appendix Course Descriptions from Oslo for courses descriptions.

Text Books

The lack of textbooks was initially a problem, but three books form the core of the principles of the course. The first book newcomers need is Mark Burgess' *Principles of Network and System Administration* [6]. Without this book, students who have

never worked as system administrators have no idea what the subject is about. Later, the *Practice of System and Network Administration* by Limoncelli and Hogan [14] provides excellent advice about experiential and management matters, but remains difficult for students to understand without experience of working as a system administrator. It is recommended as supporting literature as it is eminently readable. Finally, we use *Analytical Network and System Administration* along with extensive exercises as the basis for teaching scientific method. This book is too difficult in its presentation however and only excerpts are used in practice. In addition to these core books, we provide students with a library of many technical books on special topics.

Finally, to bring system administration up to the state of the art, we have collaborated in the publication of a new collection of essays by experts in the field of system administration called the *Handbook of Network and System Administration* [10].

Examination Forms

We require students to read, write, speak and "do" well during their sojourn at the College. The ability to both think and communicate those thoughts is central to our ethos. We test students on their skills in

- Impartial reporting of procedure and results (scientific method).
- Opinion or standpoint formulation (decision-making).

Our preferred examination form is the oral examination combined with course work. This is a very cost-effective approach to gauging students' understanding, as long as student numbers are not too great. By teaching at Masters level, we can make this economically viable to keep a smaller number of students, supported by research activity.

Even students who are initially weak in English language end up being able to make reasonable presentations and have no serious problems in mastering this form of examination, thanks to a consistent emphasis on communication skills throughout the course.

Relationships Between Courses

The course programme was designed with particular care (see Figure 1) to teach concepts, theory and combine this practical experience. Since some concepts require skills that computer scientists are particularly poor in (e.g., calculus, statistics and empiricism) it is especially important to introduce these concepts repeatedly over time. The programme is composed of four semesters which have principal goals as follows:

1. Provide introductory or fundamental concepts, knowledge and skills.
2. To make students independent in their learning, experience self-learning by trial and error etc.
3. To teach students how to view the world analytically using models and experimentation followed by interpretation. Students are encouraged to develop the capacity for original thought.

4. All of the above are implemented in an individual project.

Although Masters degree projects are increasingly being allowed as group efforts in colleges and universities, we strongly encourage students to undertake individual projects. In certain cases projects have been related, but then each student writes an independent thesis.

The figure shows how courses follow on from one another and build knowledge and experience that is reused in later courses. The 'most important' courses are those which collate, integrate or serve knowledge to and from the largest number of other courses. These include the laboratory course and the analytical methods course. Naturally the final thesis itself essentially builds on all of the foregoing courses. However, this depends on the exact nature of the project chosen by the student.

In addition to actual subject matter, there are several cultural delineations of the same material by different industrial and academic subgroups:

- Network Operations community perspective.
- The telecom industry perspective.
- The UNIX community perspective.
- The Microsoft Windows or Apple MacOS perspective.

Although we try to cover all of these viewpoints, with emphasis depending on context, we are notably stronger in some areas than others. In particular our key strengths at OUC are in UNIX. We have found that it is useful to maintain this emphasis in our courses since it makes our curriculum unique, and fills a niche in education that is missing from curricula of other institutions, where Microsoft systems are often covered exclusively.

One Year Amsterdam Programme

The one-year master's degree programme in System and Network Engineering (SNE) is a successful

new study programme provided by the Universiteit van Amsterdam (UvA). It is offered in collaboration with the Hogeschool van Amsterdam (HvA). In 2003, an enthusiastic team of experienced system and network administrators with academic backgrounds initiated the process of designing and implementing this degree programme.

Almost immediately, it became apparent that the programme served a previously unmet societal function: the provision of academically trained system and network administrators. In the past years, it has also become obvious that students who have completed the programme successfully have found positions in a wide variety of areas. Nearly without exception, the employers concerned have commented on the valuable knowledge and skills that these students have brought to their work.

This success is partially the result of the programme's design, organisation and philosophy, with special considerations related to the short one-year track.

The following are among the valuable features of the degree programme:

- A strong connection to practice with an approach that is simultaneously theoretical, product-independent and supplier-independent.
- A solid, clearly outlined and well-coordinated programme.
- Strong social cohesion resulting from the large amount of time that students spend together in the SNE Lab.
- Strong motivation and an excellent attitude toward study on the part of the students, partially due to a strict admission procedure.
- Emphasis on concepts (knowledge/insight), with less emphasis on operational procedures.
- A unique design for the final phase of the programme, which is divided into two short (one-month) research projects.

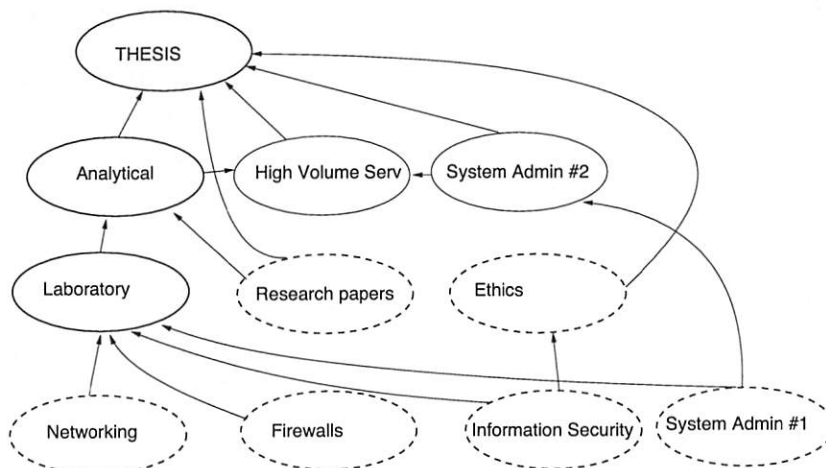


Figure 2: Dotted courses are fundamental sources of information and skills. Heavy lines show the most 'central' courses, i.e., those with the greatest connectivity or impact on collating knowledge. The four levels correspond approximately to the four semesters.

- Considerable emphasis on the preparation of reports and presentations.
- Emphasis on current relevant themes, including open technology and security.
- Strong involvement on the part of (core) lecturers.
- Continuous evaluation and adjustment.
- Strong connection to research, as conducted by the SNE group of the UvA.

At the outset of the programme the name System and Network Administration was chosen. Experiences with all of the classes that have thus far completed the programme and with the employers who have hired the graduates have shown that, in practice, the concept of system administration is not usually associated with academic training. Therefore the name of the programme (but not its contents) has been changed to System and Network Engineering. Engineering is a better *pars pro toto*, because the programme places considerable emphasis on the architecture of systems and networks, including the engineering aspects, besides administration and management.

For more information about the structure and organisation of the Amsterdam SNE master we refer to the self-assessment made for the accreditation committee visiting in March 2007 [15].

Goals

The Amsterdam education has the same general goals as the Oslo master as detailed in Goals. Some more specific qualifications can be specified as follows:

1. Graduates should be skilled in exploring (searching, reading and evaluating) the many forms of documentation and literature concerning system and network engineering, with regard to both content and medium. They should be familiar with the ISOC, the W3C, IEEE and other international bodies that develop standards and publish in the area of computer systems and networks.
2. Graduates should be very familiar with the usual configurations and procedures for the normal and crisis administration of a variety of current systems and networks, middleware and applications. They should therefore be quickly employable in the usual multi-vendor systems and network contexts.
3. Graduates should be very familiar with the security functions of systems and networks, and they should be capable of contributing actively to the architecture and configuration of systems and networks that conform to current security standards. Graduates should also be able to determine whether systems or networks conform to particular security standards.
4. Graduates should have the technical knowledge of communication protocols, network components and business systems that they will need to accurately justify choices and steps relating to

administration and security, including those regarding configuration, procedures and security architecture.

5. Graduates should have sufficient insight into the organisational contexts within which systems and networks function to channel the needs of organisations and users, and to translate them into appropriate technical support.
6. Graduates should have sufficient technical knowledge and intellectual capacity to assume positions of leadership in the field of system and network engineering within a few years. They should have the capacity to develop their own vision of the field of system and network engineering, thus contributing to evolution and innovation in concrete system environments.

Relationship Between Courses

In an effort to realise a coherent programme, two main topics were formulated to serve as a unifying theme for the entire study programme. These topics relate to current and important themes within the profession.

- Open Technology, which involves open standards (e.g., RFCs), open software (including open source) and open security (the antithesis of security through obscurity).
- Security, both technical and non-technical.

With respect to the educational design, the degree programme is based upon a set curriculum, which is the same for all students. The courses are offered according to a set schedule with many contact hours and an attendance requirement. The programme includes four technical-theoretical courses (CIA,² SSN, INR and LIA), one non-technical theoretical course (ICP), one 'basics' course (ESA), two practical courses (DIA, IDS) and two research projects (RP1, RP2). The interdependence of these courses is depicted in Figure 3.

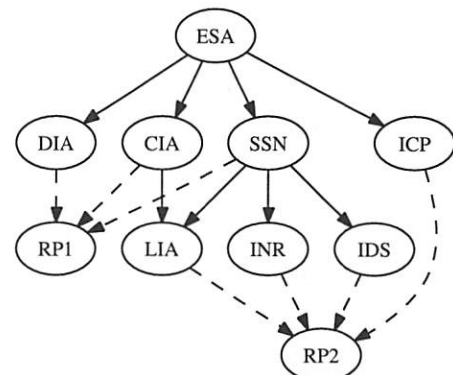


Figure 3: Interdependence of SNE courses.

This approach, according to which students follow largely the same programme, strengthens the coherence of the programme and the commitment of the

²For an explanation of the course abbreviations, see appendix Course Descriptions from Amsterdam.

students to the study programme. It encourages students to attend and participate in the study components, and it has a positive effect on academic achievement.

The programme is continuously evaluated through discussions with students and through regular meetings by the lecturers, thereby improving the coherence of the various components.

The curriculum involves a gradually increasing level of difficulty and during the course of the year students work more and more independently. This effect can be clearly observed in the research projects, within which the quality of the work that is submitted at the end of the year is clearly superior to that of similar work that was submitted halfway through the programme.

Programme Comparison

It is instructive to compare the approaches in the two programmes carried out for the university Accreditation committee for the master education SNE at the Universiteit van Amsterdam. We reproduce their table here, see Table 1, showing the correspondence. It is interesting to see that both courses emphasize the same basic ideas, even though the mode of implementation is somewhat different.

Course Implementations

Oslo

The implementation of any new programme presents a number of challenges both in terms of resources and imagination. We have now seen the progress of four groups of students at our college and are able to draw a number of experiences from this time. The opportunity to craft the entire curriculum without interference from national academic title regulation has also made the course curriculum highly cohesive and integrated in comparison to the often poorly cohesive topics at Bachelor level. Students notice

this difference and have commented on it on several occasions in evaluation meetings.

The strong international nature of our group, both in the staff and the composition of the students, makes an attractive milieu for our students. We have used teaching staff from the UK, Norway, Somalia, USA, and Switzerland including our fixed members. Our students come from all over the world, including Asia and the Far East, Africa, America, and Europe. Our English speaking environment works for the most part well, and the support from the admissions office has been excellent. Our active programme of Internationalization has brought visitors from America, The Netherlands, and a number of countries participating in our European Union Network of Excellence EMAN-ICS.

Our college is especially strong in the area of UNIX and GNU/Linux technology. This gives our students a clear advantage in service management as many server room installations are still based on UNIX technology. We have chosen to maintain this focus rather than dilute it with 'more common' Microsoft technology because it gives the students an advantage. They can learn the Microsoft technologies themselves based on their UNIX knowledge. We aim rather to show how to integrate these different technologies.

A weakness of our course initially was that it did not cater well to students who lacked the assumed prerequisite experience. Such ideal students were near impossible to recruit however, and so we have adapted the initial part of the course to arrange for essential background skills to be taught. The students we typically attract:

- Are looking for a way to work with computers that is not about programming.
- Have rarely any background in UNIX.
- Increasingly have not worked as system administrators either, but are attracted by the term

Oslo	ECTS	Amsterdam	ECTS
Computer and Information Security	10	Security of Systems and Networks	6
Networking: Technologies and Principles	10	InterNetworking and Routing	6
Intrusion Detection and Firewalls	10	Intrusion Detection Systems	6
Network Infrastructure and Security Laboratory	15	Assignments	
Research papers	5	Assignments	
Optional subjects	10	Distributed Internet Applications	6
		Essential Skills for Administrators	6
Analytical System Administration	10	Large Installation Administration	6
Network and System Administration #2	10	Classical Internet Applications	6
Social Aspects of Systems and Ethics	10	ICT and Company Practice	6
Final Thesis	30	Research Projects 1 and 2	6/6
	120		60

Table 1: Comparison of emphases between Oslo and Amsterdam masters courses. ECTS are European standard study points, denoting course length.

“Network” as made popular by the spread of the Internet and broadband.

One of the major strengths of our programme is a strong course in applied scientific modeling, not merely philosophy of science, but analysis, understanding and application of mathematical representations. All students are sceptical of the analytical, mathematical and scientific content we expect in the course, but many if not most students later claim that it ends up as being one of the most valuable courses to them, teaching them how to think about problems in a way that they had never learned before during their university careers.

In a recent yearly meeting of the steering council, a student representative commented that he was learning more in our programme than he ever had before. When asked why, he replied: “Because we have to.” The students regard our courses as demanding, and they see this as a positive quality.

A weakness we have identified in our course is the lack of realistic experience in the management of computer systems by the students. This is something that we have striven to simulate but without much success. Limitations in laboratory space and user activity have proven to be insurmountable problems for us. One way around this problem is to use *internships* where students spend their summer working in a local company. The practice of internships has not been widespread in Norway however, as the Norwegian summer holiday culture usually means that almost no one is left in companies in the summer months to provide guidance for students, and hence none are taken on. However, most recently we have made some progress in this area too as Norwegian society adapts.

Amsterdam

As mentioned in section on the relationship between courses, the setup of the master programme is a coherent effort centered around the themes of Open Technology and Security. “System” topics and “Network” topics are considered to be of equal importance and are treated as a unified duality throughout the year. The program is completely fixed – apart from the obvious choices in subject for the research projects – and courses are not accessible for students outside the programme. This didactic concept creates a very tight social relationship between all students and is the basis for a kind of melting pot in which teachers and students together get the maximum possible result from the efforts put into the programme. It is the firm belief of the educators in Amsterdam that such a concept is indeed necessary to be able to graduate within only one year at a sufficiently academic level, entering from a higher professional education level. The didactic concept alone is not sufficient to succeed. Students also need to belong to the top layer of the higher professional education and bring along an excellent motivation to be able to cope with the demanding year, both in time as in energy to be invested.

A major contribution to the melting pot is the so-called SNE Lab. This is a room suited for multiple purposes, accommodating about 20 students with a lecture room for centralised teaching, a desktop environment for each student, a dedicated network for production and experiments, a dedicated VM-based server for each student for projects and experiments and some space for social activities. A major part of the total study time is spent in the SNE Lab, also contributing to better graduation rates and opportunities to help each other out in case of problems.

As is the case in Oslo, students do not always have the prerequisite experience. Unconditional acceptance into the master’s degree programme in SNE is possible for students that have successfully completed one of the university bachelor’s degree programmes in Information Science. Admission into our vocationally oriented master’s degree programme is also possible for students that have successfully completed a programme at an institution of higher professional education (HBO), on condition that they pass an extensive intake examination or assessment. In practice, the majority of applications appear to come from students who have completed HBO degree programmes. The following components are part of the intake procedure:

General Skills

- Literature skills: reading and summarising a technical document.
- Oral skills: presenting a previously written thesis.
- Analytical skills: basic knowledge of discrete mathematics.

Specific Skills

- Basic knowledge of UNIX.
- Basic network knowledge (TCP/IP).
- Basic knowledge of scripting (shell).

Within the actual programme, the ESA course offers students the opportunity to brush up on the knowledge and skills that they will need to be able to follow the rest of the programme successfully.

The master does not result in a regular thesis project. The one-year format demands another construction here. Most students have already written a thesis for their bachelor’s. The competence added for the SNE master is the ability to research an academic topic in a very short time (one month) and write a consultancy report about it. All learned skills and knowledge come together within these projects as far as researching, communicating, writing and presenting is concerned. A disadvantage of this setup is that the scope of research has to be restricted to fit within the one month period. An advantage is that producing timely and short reports giving insight into a problem is of great practical advantage in real life situations. Moreover this format creates room to exercise these skills twice a year (in January and in June).

An often heard remark made by students is that they learn in a short time more in our education than

in years before in other educations. This is almost certainly due to the didactic concept and also shows that students are able to perform better under the right circumstances.

Our education is in Dutch, because we do not aim at recruiting international students right now. A larger group than 20 students would also put stress on the didactic concept. All teaching material however is in English and a thorough understanding of English in listening, reading and writing is necessary. Most study material is presented in the form of slides, accompanying the lectures, or is available online on the Internet. For some courses we make use of books of renowned authors in the subject field like [16, 17, 18].

Employment of the Students

Oslo

Our former students have worked in a variety of locations. A few of these are listed below.

- Norwegian Cancer Research Fund
- Opera Software
- IBM Norway
- The Armed Forces Computer Facility
- Oslo university Computer Operations Center
- Telia
- Norsk Hydro
- Oslo Data Center (ODS)
- Our group (as system administrator and teacher)
- Freecode
- Linpro AS

Amsterdam

Our alumni have a very good prospect as much wanted professionals in a variety of employment opportunities. To name a few:

- Continued research in a Ph.D. position
- Consultancy in small and large advising firms
- Lecturer in higher education
- Network Operations with ISPs and big infrastructure providers
- ICT management in small and large companies
- System architect and engineer
- Security specialist and auditor

Summary

This paper is a statement of impressions. It is clearly too early to make any scientific evaluation of the success of our experiments. Nonetheless we are confident enough in our results to report on our activities and encourage others to follow suit. The empirical quality of our reporting will improve as more years of experience can be collated. With four year of experience we can only assert that, subject to minor adjustments, our programmes have succeeded well in their general goals, but that there is still room for improvement in the details.

We struggle occasionally with resources, but we have identified a strategy for rationalizing teaching burdens by expanding the curriculum marginally.

We continue to monitor and discuss the results of our programmes between our universities, making adjustments on the fly. We believe that these programmes have been a success not only in terms of numbers of students emerging from the conveyor belt, but also in generating an academic cohesion within our groups, stimulating research, and attracting doctoral students, post-docs and international visitors who have enriched our computer science milieu for everyone.

Acknowledgement

We would like to thank the other staff members at our institutions who are involved in teaching and developing the materials for these courses. Naturally we also thank the many talented students who have passed through our corridors and made the teaching worth-while. Mark would like to acknowledge Steve VanDevender for interesting discussions about system administration education that helped to motivate this paper. MB is supported by the EC IST-EMANICS Network of Excellence (#26854).

Appendix: Course Descriptions from Oslo Network and System Administration I

The aim of this course is to provide an understanding of the role and procedures carried out by network and system administrators. It outlines general principles, while providing concrete hands-on examples using [6].

Computer and Information Security

An introduction to the theory and concepts of security, as applied to computer systems. The course builds on the earlier course on system administration, where security was discussed from a practical viewpoint. A deeper understanding of the principles and examples of security is explored, going beyond the encryption style of many courses on security. Common problems in software design are also noted.

Networking: Technologies and Principles

Explaining the principles and practice of networking technologies and protocols that are used today, for transferring and routing information between hosts. Analysis is a key part of understanding networking, and students are expected to develop a sufficient understanding of the physics and mathematics of communication and traffic flow to gain full marks.

The course introduces a theoretical foundation for the coming laboratory course, and the exercises teach some practical and diagnostic skills in using these protocols, including tools such as traceroute, and the router operating systems IOS/JunOS.

Intrusion Detection and Firewall Security

This course introduces the fundamentals of TCP/IP network monitoring at the packet level and its application to computer security. Detecting and protecting against hostile network activity has become

one of the important topics of Network and System Administration.

Social and Ethical Aspects of Systems and Research

Originally two separate courses, this module now consists of two courses under one umbrella. In both these modules students are expected to read and write about system administration. In research papers, they read the academic literature published in the field and summarize it. This is an important skill for the master thesis and for comprehension in a professional setting. In the Social and Ethical component, students are expected to formulate a standpoint of their own and argue for it. This develops communication skills, comprehension skills and powers of rational judgment.

Network Infrastructure and Security Lab

The aim of this course is to give students substantial experience in using networking and system administration equipment in as realistic an environment as we can create. Students should become proficient at handling hardware and software, as well as learn debugging skills and scientific methodology. Experiments carried out in the lab must be documented as scientific experiments and written up in clear, concise English. Presentation skills contribute to the grade, as well as documentation of analytical ability and systematic work practice. Students are graded on performance, reporting, tidiness and safety.

Network and System Administration II

This is a course in essential services and software tools and application services such as DNS, NFS, SMB etc. It includes understanding integration of Microsoft and Macintosh OS and network models, evaluating and gaining experience of data backup and archiving models. Directory Services, Database Administration, automation of maintenance of a LAN with Cengine are covered. Economical aspects of system management in planning and resource management are discussed.

Analytical Methods for Systems

The aim of this course is to provide a deeper perspective on systems and their administration from a theoretical and cultural viewpoint. Only such a background can stimulate research and creative solutions in the future. The course places system administration in the contexts of other subjects. Students must master some basic modeling techniques and be able to apply simple mathematical methods to problems of planning and analysis of human-computer systems. The results are applied to business process modeling also.

High Volume Computing Services

The aim of this course is to cover the fundamentals of implementing scalable computing systems for parallel computation and service delivery. This includes an understanding of the science of data centers where large installations are kept, and understanding the

meaning of scalability. Students should know the difference between High Performance Computing, High Availability Computing and High Volume Computing and know how Amdahl's law applies to parallelization "speedup" and load balancing. Some basic queueing theory is learned.

Supercomputers and Virtual Operating Systems

This course is designed to fill a specific need for the computing industry: experience and understanding of supercomputers and virtual operating systems. Specific attention is given the IBM operating systems zOS (OS/390) and VM that are widely deployed in finance and banking sectors but which are rarely covered in university curricula.

Final Thesis

A one semester project, planned and executed by the student.

Course Descriptions from Amsterdam

Essential Skills for Administrators (ESA)

This course forms the foundation for much of the daily work of a system administrator. If the use of open standards and open source software is to be advocated credibly, system administrators must adhere to these standards as well. In the area of documentation, attention is paid to (pdf)(La)TeX, and XHTML is considered for Web purposes. Version tools, including RCS, CVS and SVN are also addressed, as is the use of secure remote log-in (SSH) and secure communication (PGP, GPG). Finally, a number of scripting languages (shell, Perl, Python, Tcl/Tk and Ruby) are discussed. Written reporting is an important component of the course.

Classical Internet Applications (CIA)

The aim of the course is to understand basic architectural issues in classical client-server environments. Topics covered are:

- Historical awareness of the development of Internet and UNIX.
- Insight into and knowledge of the most important classical client-server applications (DNS, Email, Web and Directory Services).
- Understanding the role of security in designing systems that must carry out the identified services.

Security of Systems and Networks (SSN)

Systems are secured according to a variety of principles, including plain-text passwords, one-time passwords, encrypted passwords, public/private keys and certificates. Networks are secured with firewalls and encryption on the network layer. The topics that are addressed in this course include remote access using SSH, secure Web transactions using SSL/TLS, single sign-on using Kerberos, secure email using PGP/GPG, IPsec and key management. The course also considers the problems of wireless access and WEP. Many of the

security systems that are mentioned for these purposes are based on the encoding of information (encryption). The course also addresses the (mathematical) principles of cryptography. The following skills receive special emphasis:

- Evaluation of security technology.
- Cooperation in groups of two and four.
- Independent literature searches.
- Written reporting.

Distributed Internet Applications (DIA)

The issues concerning the development of middleware systems for large-scale computer networks are discussed. Principles taught include communication, processes, naming, consistency and replication, fault tolerance, and security. These principles are further explained by means of different paradigms applied to distributed systems: object-based systems, distributed file systems (NFS), document-based systems (the Web), and coordination-based systems (publish/subscribe systems). Explicit attention is paid to the practical feasibility and scalability of various solutions. For this reason, experimental (research) systems as well as commercially available systems are discussed.

InterNetworking and Routing (INR)

This course is all about the world of ISPs and transport providers in the Internet. Topics discussed are:

- Mathematical modeling of addressing and routing in the Internet (both IPv4 and IPv6).
- Insight into and knowledge of the abstract and concrete algorithms that are used in routing systems.
- Insight into and knowledge of the virtualisation techniques that can be used to study networks and their routing systems (focusing on RIP, OSPF and BGP).

Large Installation Administration (LIA)

The daily tasks of system administrators and the concepts with which they should be familiar are the focus of this course, which addresses the design, implementation and documentation of procedures for daily administration. Security, stability and manageability are primary requirements in this regard. The course addresses account management, storage management and version management. Particular attention will be paid to the administration of complex systems and networks in large organisations. The course also addresses ITIL, PRINCE2 and other technologies. Finally, presentation skills are emphasised in this course as well.

Intrusion Detection Systems (IDS)

This course focuses on methods and techniques to detect anomalous behaviour, to report on it and to take appropriate measures. Topics included are:

- Examination of hacker techniques and the study of security systems from a hacker's perspective.

- Detection techniques, including Intrusion Detection Systems and Honeynets.
- Protocol analysis and knowledge of tools.
- Rootkit and malware technology.
- Cooperation in groups of three or four and submission of a proof of concept.
- Independent literature searches.
- Ethical aspects of security and network technology.
- Presentation of research results and written reporting.

ICT and Company Practice (ICP)

Master-level positions within organisations require the capacity to form and defend well-founded opinions concerning business-related ICT issues. Those who hold such positions serve as discussion partners for a wide range of managers, advisors and policy makers, many of whom are not technically oriented.

A well-founded opinion is formed through the acquisition of knowledge. The ability to defend an opinion requires good communication (by means of presentations and written reports).

The objectives of the ICP course can be formulated as follows:

- The acquisition of knowledge concerning business-oriented ICT issues. Relevant topics are sourcing, legacy and information security. These topics are chosen specifically because of their interface with the work and responsibilities of system and network administrators.
- Critical evaluation of a non-technical scientific article in the area of ICT.
- Writing a non-technical scientific article in the area of ICT.

Research Projects 1 and 2 (RP1, RP2)

The course objective is to ensure that students become acquainted with problems from the field of practice through two short projects, which require the development of non-trivial methods, concepts and solutions. Each course is a very intensive one-month full-time activity. After these courses, students should be able to:

- Transform a roughly outlined problem into a carefully defined question, supported by literature on the topic.
- Establish a feasible project schedule for answering the question.
- Conduct autonomous research to answer the question at hand, using literature searches, studying, experimentation and/or the development of software and hardware.
- Present solutions to a diverse audience (experts as well as non-experts).
- Defend solutions in debates.
- Provide an appropriate report that is useful to a client.

Author Biographies

Mark Burgess is professor of Network and System Administration at Oslo University College. He was the first professor with this title. Mark obtained a Ph.D. in Theoretical Physics in Newcastle, for which he received the Runcorn Prize. His current research interests include the behaviour of computers as dynamic systems and applying ideas from physics to describe computer behaviour. Mark is the author of the popular configuration management software package cfengine. He made important contributions to the theory of the field of automation and policy based management, including the idea of operator convergence and promise theory. He is the author of numerous books and papers on Network and System Administration and has won several prizes for his work. Reach him electronically at Mark.Burgess@iu.hio.no.

Karst Koymans holds an M.Sc. in Mathematics from Utrecht University and a Ph.D. in Mathematical Logic also from Utrecht University. His thesis is titled "Models of the Lambda Calculus." Karst Koymans has been working as an Assistant and Associate Professor at Utrecht University and the University of Amsterdam. During that period he gained extensive experience with System and Network Engineering, mostly by being network and system administrator for his faculty. Since 2003 he is the director of the master education System and Network Engineering at the University of Amsterdam.

Bibliography

- [1] Herzog, U., "Network Planning and Performance Engineering," *Network and Distributed Systems Management (Edited by M. Sloman)*, p. 349, 1994.
- [2] IFIP/IEEE Conference, *Network Operations and Management Symposium*.
- [3] IFIP/IEEE Conference, *Integrated Network Management*.
- [4] IARIA Conferences, (various), <http://www.iaria.org>.
- [5] IFIP/IEEE Conference, *Distributed Systems Operations and Management*.
- [6] Burgess, M., *Principles of Network and System Administration*, J. Wiley & Sons, Chichester, 2000.
- [7] Burgess, M., *Analytical Network and System Administration – Managing Human-Computer Systems*, J. Wiley & Sons, Chichester, 2004.
- [8] Hasle, Thor E., *Nettverksadministrasjon, 2nd Edition*, Cappelen Akademisk Forlag, Oslo, 2003.
- [9] ACM College Education Committee, *Guidelines for Associate-Degree Programs to Support Computing in a Networked Environment*, Technical Report, 2000.
- [10] Bergstra, J. and M. Burgess, editors, *The Handbook of Network and System Administration*, Elsevier, 2007.
- [11] Burgess, M. and G. Undheim, "Predictable Scaling Behaviour in the Data Center with Multiple Application Servers," *Lecture Notes on Computer Science, Proceedings 17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)*, Vol. 4269, pp. 9-60, Springer, 2006.
- [12] Burgess, M. and S. I. Ulland, "Uncertainty in Global Application Services with Load Sharing," *Lecture Notes on Computer Science, Proc. 17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)*, Vol. 4269, pp. 37-48, Springer, 2006.
- [13] Björnstad, J. H. and M. Burgess, "On the Reliability of Service Level Estimators in the Data Center," *Proceedings 17th IFIP/IEEE Integrated Management*, submitted, Springer, 2006.
- [14] Limoncelli, T. and C. Hogan, *The Practice of System and Network Administration*, Addison Wesley, 2003.
- [15] Koymans, C. P. J., et al., "Self-Assessment of the Degree Course System and Network Administration," Educational Institute Information Sciences, Universiteit van Amsterdam, https://www.os3.nl/_media/2006-2007/accreditatie.pdf.
- [16] Speciner, Mike, Charlie Kaufman, Radia Perlman, *Network Security: Private Communication in a Public World, 2/E*, Prentice Hall, 2003.
- [17] Steen, Maarten van, Andrew S. Tanenbaum, *Distributed Systems: Principles and Paradigms, 2/E*, Prentice Hall, 2007.
- [18] Perlman, Radia, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols, 2nd Edition*, Addison-Wesley Professional, 2000.

On Designing and Deploying Internet-Scale Services

James Hamilton – Windows Live Services Platform

ABSTRACT

The system-to-administrator ratio is commonly used as a rough metric to understand administrative costs in high-scale services. With smaller, less automated services this ratio can be as low as 2:1, whereas on industry leading, highly automated services, we've seen ratios as high as 2,500:1. Within Microsoft services, Autopilot [1] is often cited as the magic behind the success of the Windows Live Search team in achieving high system-to-administrator ratios. While auto-administration is important, the most important factor is actually the service itself. Is the service efficient to automate? Is it what we refer to more generally as operations-friendly? Services that are operations-friendly require little human intervention, and both detect and recover from all but the most obscure failures without administrative intervention. This paper summarizes the best practices accumulated over many years in scaling some of the largest services at MSN and Windows Live.

Introduction

This paper summarizes a set of best practices for designing and developing operations-friendly services. This is a rapidly evolving subject area and, consequently, any list of best practices will likely grow and morph over time. Our aim is to help others

1. deliver operations-friendly services quickly and
2. avoid the early morning phone calls and meetings with unhappy customers that non-operations-friendly services tend to yield.

The work draws on our experiences over the last 20 years in high-scale data-centric software systems and internet-scale services, most recently from leading the Exchange Hosted Services team (at the time, a mid-sized service of roughly 700 servers and just over 2.2M users). We also incorporate the experiences of the Windows Live Search, Windows Live Mail, Exchange Hosted Services, Live Communications Server, Windows Live Address Book Clearing House (ABCH), MSN Spaces, Xbox Live, Rackable Systems Engineering Team, and the Messenger Operations teams in addition to that of the overall Microsoft Global Foundation Services Operations team. Several of these contributing services have grown to more than a quarter billion users. The paper also draws heavily on the work done at Berkeley on Recovery Oriented Computing [2, 3] and at Stanford on Crash-Only Software [4, 5].

Bill Hoffman [6] contributed many best practices to this paper, but also a set of three simple tenets worth considering up front:

1. Expect failures. A component may crash or be stopped at any time. Dependent components might fail or be stopped at any time. There will be network failures. Disks will run out of space. Handle all failures gracefully.
2. Keep things simple. Complexity breeds problems. Simple things are easier to get right. Avoid unnecessary dependencies. Installation

should be simple. failures on one server should have no impact on the rest of the data center.

3. Automate everything. People make mistakes. People need sleep. People forget things. Automated processes are testable, fixable, and therefore ultimately much more reliable. Automate wherever possible.

These three tenets form a common thread throughout much of the discussion that follows.

Recommendations

This section is organized into ten sub-sections, each covering a different aspect of what is required to design and deploy an operations-friendly service. These sub-sections include overall service design; designing for automation and provisioning; dependency management; release cycle and testing; hardware selection and standardization; operations and capacity planning; auditing, monitoring and alerting; graceful degradation and admission control; customer and press communications plan; and customer self provisioning and self help.

Overall Application Design

We have long believed that 80% of operations issues originate in design and development, so this section on overall service design is the largest and most important. When systems fail, there is a natural tendency to look first to operations since that is where the problem actually took place. Most operations issues, however, either have their genesis in design and development or are best solved there.

Throughout the sections that follow, a consensus emerges that firm separation of development, test, and operations isn't the most effective approach in the services world. The trend we've seen when looking across many services is that low-cost administration correlates highly with how closely the development, test, and operations teams work together.

In addition to the best practices on service design discussed here, the subsequent section, “Designing for Automation Management and Provisioning,” also has substantial influence on service design. Effective automatic management and provisioning are generally achieved only with a constrained service model. This is a repeating theme throughout: simplicity is the key to efficient operations. Rational constraints on hardware selection, service design, and deployment models are a big driver of reduced administrative costs and greater service reliability.

Some of the operations-friendly basics that have the biggest impact on overall service design are:

- **Design for failure.** This is a core concept when developing large services that comprise many cooperating components. Those components will fail and they will fail frequently. The components don’t always cooperate and fail independently either. Once the service has scaled beyond 10,000 servers and 50,000 disks, failures will occur multiple times a day. If a hardware failure requires any immediate administrative action, the service simply won’t scale cost-effectively and reliably. The entire service must be capable of surviving failure without human administrative interaction. Failure recovery must be a very simple path and that path must be tested frequently. Armando Fox of Stanford [4, 5] has argued that the best way to test the failure path is never to shut the service down normally. Just hard-fail it. This sounds counter-intuitive, but if the failure paths aren’t frequently used, they won’t work when needed [7].

- **Redundancy and fault recovery.** The mainframe model was to buy one very large, very expensive server. Mainframes have redundant power supplies, hot-swappable CPUs, and exotic bus architectures that provide respectable I/O throughput in a single, tightly-coupled system. The obvious problem with these systems is their expense. And even with all the costly engineering, they still aren’t sufficiently reliable. In order to get the fifth 9 of reliability, redundancy is required. Even getting four 9’s on a single-system deployment is difficult. This concept is fairly well understood industry-wide, yet it’s still common to see services built upon fragile, non-redundant data tiers.

Designing a service such that any system can crash (or be brought down for service) at any time while still meeting the service level agreement (SLA) requires careful engineering. The acid test for full compliance with this design principle is the following: is the operations team willing and able to bring down any server in the service at any time without draining the work load first? If they are, then there is synchronous redundancy (no data loss), failure

detection, and automatic take-over. As a design approach, we recommend one commonly used approach to find and correct potential service security issues: security threat modeling. In security threat modeling [8], we consider each possible security threat and, for each, implement adequate mitigation. The same approach can be applied to designing for fault resiliency and recovery.

Document all conceivable component failures modes and combinations thereof. For each failure, ensure that the service can continue to operate without unacceptable loss in service quality, or determine that this failure risk is acceptable for this particular service (e.g., loss of an entire data center in a non-geo-redundant service). Very unusual combinations of failures may be determined sufficiently unlikely that ensuring the system can operate through them is uneconomical. Be cautious when making this judgment. We’ve been surprised at how frequently “unusual” combinations of events take place when running thousands of servers that produce millions of opportunities for component failures each day. Rare combinations can become commonplace.

- **Commodity hardware slice.** All components of the service should target a commodity hardware slice. For example, storage-light servers will be dual socket, 2- to 4-core systems in the \$1,000 to \$2,500 range with a boot disk. Storage-heavy servers are similar servers with 16 to 24 disks. The key observations are:
 1. large clusters of commodity servers are much less expensive than the small number of large servers they replace,
 2. server performance continues to increase much faster than I/O performance, making a small server a more balanced system for a given amount of disk,
 3. power consumption scales linearly with servers but cubically with clock frequency, making higher performance servers more expensive to operate, and
 4. a small server affects a smaller proportion of the overall service workload when failing over.
- **Single-version software.** Two factors that make some services less expensive to develop and faster to evolve than most packaged products are
 - the software needs to only target a single internal deployment and
 - previous versions don’t have to be supported for a decade as is the case for enterprise-targeted products.

Single-version software is relatively easy to achieve with a consumer service, especially one

provided without charge. But it's equally important when selling subscription-based services to non-consumers. Enterprises are used to having significant influence over their software providers and to having complete control over when they deploy new versions (typically slowly). This drives up the cost of their operations and the cost of supporting them since so many versions of the software need to be supported.

The most economic services don't give customers control over the version they run, and only host one version. Holding this single-version software line requires

1. care in not producing substantial user experience changes release-to-release and
 2. a willingness to allow customers that need this level of control to either host internally or switch to an application service provider willing to provide this people-intensive multi-version support.
- **Multi-tenancy.** Multi-tenancy is the hosting of all companies or end users of a service in the same service without physical isolation, whereas single tenancy is the segregation of groups of users in an isolated cluster. The argument for multi-tenancy is nearly identical to the argument for single version support and is based upon providing fundamentally lower cost of service built upon automation and large-scale.

In review, the basic design tenets and considerations we have laid out above are:

- design for failure,
- implement redundancy and fault recovery,
- depend upon a commodity hardware slice,
- support single-version software, and
- enable multi-tenancy.

We are constraining the service design and operations model to maximize our ability to automate and to reduce the overall costs of the service. We draw a clear distinction between these goals and those of application service providers or IT outsourcers. Those businesses tend to be more people intensive and more willing to run complex, customer specific configurations.

More specific best practices for designing operations-friendly services are:

- **Quick service health check.** This is the services version of a build verification test. It's a sniff test that can be run quickly on a developer's system to ensure that the service isn't broken in any substantive way. Not all edge cases are tested, but if the quick health check passes, the code can be checked in.
- **Develop in the full environment.** Developers should be unit testing their components, but should also be testing the full service with their component changes. Achieving this goal efficiently requires single-server deployment (section 2.4), and the preceding best practice, a quick service health check.

- **Zero trust of underlying components.** Assume that underlying components will fail and ensure that components will be able to recover and continue to provide service. The recovery technique is service-specific, but common techniques are to
 - continue to operate on cached data in read-only mode or
 - continue to provide service to all but a tiny fraction of the user base during the short time while the service is accessing the redundant copy of the failed component.
- **Do not build the same functionality in multiple components.** Foreseeing future interactions is hard, and fixes have to be made in multiple parts of the system if code redundancy creeps in. Services grow and evolve quickly. Without care, the code base can deteriorate rapidly.
- **One pod or cluster should not affect another pod or cluster.** Most services are formed of pods or sub-clusters of systems that work together to provide the service, where each pod is able to operate relatively independently. Each pod should be as close to 100% independent and without inter-pod correlated failures. Global services even with redundancy are a central point of failure. Sometimes they cannot be avoided but try to have everything that a cluster needs inside the clusters.
- **Allow (rare) emergency human intervention.** The common scenario for this is the movement of user data due to a catastrophic event or other emergency. Design the system to never need human interaction, but understand that rare events will occur where combined failures or unanticipated failures require human interaction. These events will happen and operator error under these circumstances is a common source of catastrophic data loss. An operations engineer working under pressure at 2 a.m. will make mistakes. Design the system to first not require operations intervention under most circumstances, but work with operations to come up with recovery plans if they need to intervene. Rather than documenting these as multi-step, error-prone procedures, write them as scripts and test them in production to ensure they work. What isn't tested in production won't work, so periodically the operations team should conduct a "fire drill" using these tools. If the service-availability risk of a drill is excessively high, then insufficient investment has been made in the design, development, and testing of the tools.
- **Keep things simple and robust.** Complicated algorithms and component interactions multiply the difficulty of debugging, deploying, etc. Simple and nearly stupid is almost always better in a high-scale service-the number of interacting failure modes is already daunting before complex optimizations are delivered. Our general rule is that optimizations that bring an

order of magnitude improvement are worth considering, but percentage or even small factor gains aren't worth it.

- **Enforce admission control at all levels.** Any good system is designed with admission control at the front door. This follows the long-understood principle that it's better to not let more work into an overloaded system than to continue accepting work and beginning to thrash. Some form of throttling or admission control is common at the entry to the service, but there should also be admission control at all major components boundaries. Work load characteristic changes will eventually lead to sub-component overload even though the overall service is operating within acceptable load levels. See the note below in section 2.8 on the "big red switch" as one way of gracefully degrading under excess load. The general rule is to attempt to gracefully degrade rather than hard failing and to block entry to the service before giving uniform poor service to all users.
- **Partition the service.** Partitions should be infinitely-adjustable and fine-grained, and not be bounded by any real world entity (person, collection ...). If the partition is by company, then a big company will exceed the size of a single partition. If the partition is by name prefix, then eventually all the P's, for example, won't fit on a single server. We recommend using a look-up table at the mid-tier that maps fine-grained entities, typically users, to the system where their data is managed. Those fine-grained partitions can then be moved freely between servers.
- **Understand the network design.** Test early to understand what load is driven between servers in a rack, across racks, and across data centers. Application developers must understand the network design and it must be reviewed early with networking specialists on the operations team.
- **Analyze throughput and latency.** Analysis of the throughput and latency of core service user interactions should be performed to understand impact. Do so with other operations running such as regular database maintenance, operations configuration (new users added, users migrated), service debugging, etc. This will help catch issues driven by periodic management tasks. For each service, a metric should emerge for capacity planning such as user requests per second per system, concurrent on-line users per system, or some related metric that maps relevant work load to resource requirements.
- **Treat operations utilities as part of the service.** Operations utilities produced by development, test, program management, and operations should be code-reviewed by development, checked into the main source tree, and tracked on the same

schedule and with the same testing. Frequently these utilities are mission critical and yet nearly untested.

- **Understand access patterns.** When planning new features, always consider what load they are going to put on the backend store. Often the service model and service developers become so abstracted away from the store that they lose sight of the load they are putting on the underlying database. A best practice is to build it into the specification with a section such as, "What impacts will this feature have on the rest of the infrastructure?" Then measure and validate the feature for load when it goes live.
- **Version everything.** Expect to run in a mixed-version environment. The goal is to run single version software but multiple versions will be live during rollout and production testing. Versions n and $n+1$ of all components need to co-exist peacefully.
- **Keep the unit/functional tests from the previous release.** These tests are a great way of verifying that version $n-1$ functionality doesn't get broken. We recommend going one step further and constantly running service verification tests in production (more detail below).
- **Avoid single points of failure.** Single points of failure will bring down the service or portions of the service when they fail. Prefer stateless implementations. Don't affinitize requests or clients to specific servers. Instead, load balance over a group of servers able to handle the load. Static hashing or any static work allocation to servers will suffer from data and/or query skew problems over time. Scaling out is easy when machines in a class are interchangeable. Databases are often single points of failure and database scaling remains one of the hardest problems in designing internet-scale services. Good designs use fine-grained partitioning and don't support cross-partition operations to allow efficient scaling across many database servers. All database state is stored redundantly (on at least one) fully redundant hot standby server and failover is tested frequently in production.

Automatic Management and Provisioning

Many services are written to alert operations on failure and to depend upon human intervention for recovery. The problem with this model starts with the expense of a 24x7 operations staff. Even more important is that if operations engineers are asked to make tough decisions under pressure, about 20% of the time they will make mistakes. The model is both expensive and error-prone, and reduces overall service reliability.

Designing for automation, however, involves significant service-model constraints. For example, some of the large services today depend upon database systems with asynchronous replication to a secondary,

back-up server. Failing over to the secondary after the primary isn't able to service requests loses some customer data due to replicating asynchronously. However, not failing over to the secondary leads to service downtime for those users whose data is stored on the failed database server. Automating the decision to fail over is hard in this case since it's dependent upon human judgment and accurately estimating the amount of data loss compared to the likely length of the down time. A system designed for automation pays the latency and throughput cost of synchronous replication. And, having done that, failover becomes a simple decision: if the primary is down, route requests to the secondary. This approach is much more amenable to automation and is considerably less error prone.

Automating administration of a service after design and deployment can be very difficult. Successful automation requires simplicity and clear, easy-to-make operational decisions. This in turn depends on a careful service design that, when necessary, sacrifices some latency and throughput to ease automation. The trade-off is often difficult to make, but the administrative savings can be more than an order of magnitude in high-scale services. In fact, the current spread between the most manual and the most automated service we've looked at is a full two orders of magnitude in people costs.

Best practices in designing for automation include:

- **Be restartable and redundant.** All operations must be restartable and all persistent state stored redundantly.
- **Support geo-distribution.** All high scale services should support running across several hosting data centers. In fairness, automation and most of the efficiencies we describe here are still possible without geo-distribution. But lacking support for multiple data center deployments drives up operations costs dramatically. Without geo-distribution, it's difficult to use free capacity in one data center to relieve load on a service hosted in another data center. Lack of geo-distribution is an operational constraint that drives up costs.
- **Automatic provisioning and installation.** Provisioning and installation, if done by hand, is costly, there are too many failures, and small configuration differences will slowly spread throughout the service making problem determination much more difficult.
- **Configuration and code as a unit.** Ensure that
 - the development team delivers the code and the configuration as a single unit,
 - the unit is deployed by test in exactly the same way that operations will deploy it, and
 - operations deploys them as a unit.

Services that treat configuration and code as a unit and only change them together are often more reliable.

If a configuration change must be made in production, ensure that all changes produce an audit log record so it's clear what was changed, when and by whom, and which servers were effected (see section 2.7). Frequently scan all servers to ensure their current state matches the intended state. This helps catch install and configuration failures, detects server misconfigurations early, and finds non-audited server configuration changes.

- **Manage server roles or personalities rather than servers.** Every system role or personality should support deployment on as many or as few servers as needed.
- **Multi-system failures are common.** Expect failures of many hosts at once (power, net switch, and rollout). Unfortunately, services with state will have to be topology-aware. Correlated failures remain a fact of life.
- **Recover at the service level.** Handle failures and correct errors at the service level where the full execution context is available rather than in lower software levels. For example, build redundancy into the service rather than depending upon recovery at the lower software layer.
- **Never rely on local storage for non-recoverable information.** Always replicate all the non-ephemeral service state.
- **Keep deployment simple.** File copy is ideal as it gives the most deployment flexibility. Minimize external dependencies. Avoid complex install scripts. Anything that prevents different components or different versions of the same component from running on the same server should be avoided.
- **Fail services regularly.** Take down data centers, shut down racks, and power off servers. Regular controlled brown-outs will go a long way to exposing service, system, and network weaknesses. Those unwilling to test in production aren't yet confident that the service will continue operating through failures. And, without production testing, recovery won't work when called upon.

Dependency Management

Dependency management in high-scale services often doesn't get the attention the topic deserves. As a general rule, dependence on small components or services doesn't save enough to justify the complexity of managing them. Dependencies do make sense when:

1. the components being depended upon are substantial in size or complexity, or
2. the service being depended upon gains its value in being a single, central instance.

Examples of the first class are storage and consensus algorithm implementations. Examples of the second class of are identity and group management systems. The whole value of these systems is that they are a

single, shared instance so multi-instancing to avoid dependency isn't an option.

Assuming that dependencies are justified according to the above rules, some best practices for managing them are:

- **Expect latency.** Calls to external components may take a long time to complete. Don't let delays in one component or service cause delays in completely unrelated areas. Ensure all interactions have appropriate timeouts to avoid tying up resources for protracted periods. Operational idempotency allows the restart of requests after timeout even though those requests may have partially or even fully completed. Ensure all restarts are reported and bound restarts to avoid a repeatedly failing request from consuming ever more system resources.
- **Isolate failures.** The architecture of the site must prevent cascading failures. Always "fail fast." When dependent services fail, mark them as down and stop using them to prevent threads from being tied up waiting on failed components.
- **Use shipping and proven components.** Proven technology is almost always better than operating on the bleeding edge. Stable software is better than an early copy, no matter how valuable the new feature seems. This rule applies to hardware as well. Stable hardware shipping in volume is almost always better than the small performance gains that might be attained from early release hardware.
- **Implement inter-service monitoring and alerting.** If the service is overloading a dependent service, the depending service needs to know and, if it can't back-off automatically, alerts need to be sent. If operations can't resolve the problem quickly, it needs to be easy to contact engineers from both teams quickly. All teams with dependencies should have engineering contacts on the dependent teams.
- **Dependent services require the same design point.** Dependent services and producers of dependent components need to be committed to at least the same SLA as the depending service.
- **Decouple components.** Where possible, ensure that components can continue operation, perhaps in a degraded mode, during failures of other components. For example, rather than re-authenticating on each connect, maintain a session key and refresh it every N hours independent of connection status. On reconnect, just use existing session key. That way the load on the authenticating server is more consistent and login storms are not driven on reconnect after momentary network failure and related events.

Release Cycle and Testing

Testing in production is a reality and needs to be part of the quality assurance approach used by all

internet-scale services. Most services have at least one test lab that is as similar to production as (affordably) possible and all good engineering teams use production workloads to drive the test systems realistically. Our experience has been, however, that as good as these test labs are, they are never full fidelity. They always differ in at least subtle ways from production. As these labs approach the production system in fidelity, the cost goes asymptotic and rapidly approaches that of the production system.

We instead recommend taking new service releases through standard unit, functional, and production test lab testing and then going into limited production as the final test phase. Clearly we don't want software going into production that doesn't work or puts data integrity at risk, so this has to be done carefully. The following rules must be followed:

1. the production system has to have sufficient redundancy that, in the event of catastrophic new service failure, state can be quickly be recovered,
2. data corruption or state-related failures have to be extremely unlikely (functional testing must first be passing),
3. errors must be detected and the engineering team (rather than operations) must be monitoring system health of the code in test, and
4. it must be possible to quickly roll back all changes and this roll back must be tested before going into production.

This sounds dangerous. But we have found that using this technique actually improves customer experience around new service releases. Rather than deploying as quickly as possible, we put one system in production for a few days in a single data center. Then we bring one new system into production in each data center. Then we'll move an entire data center into production on the new bits. And finally, if quality and performance goals are being met, we deploy globally. This approach can find problems before the service is at risk and can actually provide a better customer experience through the version transition. Big-bang deployments are very dangerous.

Another potentially counter-intuitive approach we favor is deployment mid-day rather than at night. At night, there is greater risk of mistakes. And, if anomalies crop up when deploying in the middle of the night, there are fewer engineers around to deal with them. The goal is to minimize the number of engineering and operations interactions with the system overall, and especially outside of the normal work day, to both reduce costs and to increase quality.

Some best practices for release cycle and testing include:

- **Ship often.** Intuitively one would think that shipping more frequently is harder and more error prone. We've found, however, that more

frequent releases have less big-bang changes. Consequently, the releases tend to be higher quality and the customer experience is much better. The acid test of a good release is that the user experience may have changed but the number of operational issues around availability and latency should be unchanged during the release cycle. We like shipping on 3-month cycles, but arguments can be made for other schedules. Our gut feel is that the norm will eventually be less than three months, and many services are already shipping on weekly schedules. Cycles longer than three months are dangerous.

- **Use production data to find problems.** Quality assurance in a large-scale system is a data-mining and visualization problem, not a testing problem. Everyone needs to focus on getting the most out of the volumes of data in a production environment. A few strategies are:
 - **Measurable release criteria.** Define specific criteria around the intended user experience, and continuously monitor it. If availability is supposed to be 99%, measure that availability meets the goal. Both alert and diagnose if it goes under.
 - **Tune goals in real time.** Rather than getting bogged down deciding whether the goal should be 99% or 99.9% or any other goal, set an acceptable target and then ratchet it up as the system establishes stability in production.
 - **Always collect the actual numbers.** Collect the actual metrics rather than red and green or other summary reports. Summary reports and graphs are useful but the raw data is needed for diagnosis.
 - **Minimize false positives.** People stop paying attention very quickly when the data is incorrect. It's important to not over-alert or operations staff will learn to ignore them. This is so important that hiding real problems as collateral damage is often acceptable.
 - **Analyze trends.** This can be used for predicting problems. For example, when data movement in the system diverges from the usual rate, it often predicts a bigger problem. Exploit the available data.
 - **Make the system health highly visible.** Require a globally available, real-time display of service health for the entire organization. Have an internal website people can go at any time to understand the current state of the service.
 - **Monitor continuously.** It bears noting that people must be looking at all the data every day. Everyone should do this, but make it the explicit job of a subset of the team to do this.

- **Invest in engineering.** Good engineering minimizes operational requirements and solves problems before they actually become operational issues. Too often, organizations grow operations to deal with scale and never take the time to engineer a scalable, reliable architecture. Services that don't think big to start with will be scrambling to catch up later.
- **Support version roll-back.** Version roll-back is mandatory and must be tested and proven before roll-out. Without roll-back, any form of production-level testing is very high risk. Reverting to the previous version is a rip cord that should always be available on any deployment.
- **Maintain forward and backward compatibility.** This vital point strongly relates to the previous one. Changing file formats, interfaces, logging/debugging, instrumentation, monitoring and contact points between components are all potential risk. Don't rip out support for old file formats until there is no chance of a roll back to that old format in the future.
- **Single-server deployment.** This is both a test and development requirement. The entire service must be easy to host on a single system. Where single-server deployment is impossible for some component (e.g., a dependency on an external, non-single box deployable service), write an emulator to allow single-server testing. Without this, unit testing is difficult and doesn't fully happen. And if running the full system is difficult, developers will have a tendency to take a component view rather than a systems view.
- **Stress test for load.** Run some tiny subset of the production systems at twice (or more) the load to ensure that system behavior at higher than expected load is understood and that the systems don't melt down as the load goes up.
- **Perform capacity and performance testing prior to new releases.** Do this at the service level and also against each component since work load characteristics will change over time. Problems and degradations inside the system need to be caught early.
- **Build and deploy shallowly and iteratively.** Get a skeleton version of the full service up early in the development cycle. This full service may hardly do anything at all and may include shunts in places but it allows testers and developers to be productive and it gets the entire team thinking at the user level from the very beginning. This is a good practice when building any software system, but is particularly important for services.
- **Test with real data.** Fork user requests or workload from production to test environments. Pick up production data and put it in test environments. The diverse user population of the

product will always be most creative at finding bugs. Clearly, privacy commitments must be maintained so it's vital that this data never leak back out into production.

- **Run system-level acceptance tests.** Tests that run locally provide sanity check that speeds iterative development. To avoid heavy maintenance cost they should still be at system level.
- **Test and develop in full environments.** Set aside hardware to test at interesting scale. Most importantly, use the same data collection and mining techniques used in production on these environments to maximize the investment.

Hardware Selection and Standardization

The usual argument for SKU standardization is that bulk purchases can save considerable money. This is inarguably true. The larger need for hardware standardization is that it allows for faster service deployment and growth. If each service is purchasing their own private infrastructure, then each service has to:

1. determine which hardware currently is the best cost/performing option,
2. order the hardware, and
3. do hardware qualification and software deployment once the hardware is installed in the data center.

This usually takes a month and can easily take more.

A better approach is a "services fabric" that includes a small number of hardware SKUs and the automatic management and provisioning infrastructure on which all service are run. If more machines are needed for a test cluster, they are requested via a web service and quickly made available. If a small service gets more successful, new resources can be added from the existing pool. This approach ensures two vital principles:

1. all services, even small ones, are using the automatic management and provisioning infrastructure and
2. new services can be tested and deployed much more rapidly.

Best practices for hardware selection include:

- **Use only standard SKUs.** Having a single or small number of SKUs in production allows resources to be moved fluidly between services as needed. The most cost-effective model is to develop a standard service-hosting framework that includes automatic management and provisioning, hardware, and a standard set of shared services. Standard SKUs is a core requirement to achieve this goal.
- **Purchase full racks.** Purchase hardware in fully configured and tested racks or blocks of multiple racks. Racking and stacking costs are inexplicably high in most data centers, so let the system manufacturers do it and wheel in full racks.
- **Write to a hardware abstraction.** Write the service to an abstract hardware description. Rather than

fully exploiting the hardware SKU, the service should neither exploit that SKU nor depend upon detailed knowledge of it. This allows the 2-way, 4-disk SKU to be upgraded over time as better cost/performing systems come available. The SKU should be a virtual description that includes number of CPUs and disks, and a minimum for memory. Finer-grained information about the SKU should not be exploited.

- **Abstract the network and naming.** Abstract the network and naming as far as possible, using DNS and CNAMEs. Always, always use a CNAME. Hardware breaks, comes off lease, and gets repurposed. Never rely on a machine name in any part of the code. A flip of the CNAME in DNS is a lot easier than changing configuration files, or worse yet, production code. If you need to avoid flushing the DNS cache, remember to set Time To Live sufficiently low to ensure that changes are pushed as quickly as needed.

Operations and Capacity Planning

The key to operating services efficiently is to build the system to eliminate the vast majority of operations administrative interactions. The goal should be that a highly-reliable, 24x7 service should be maintained by a small 8x5 operations staff.

However, unusual failures will happen and there will be times when systems or groups of systems can't be brought back on line. Understanding this possibility, automate the procedure to move state off the damaged systems. Relying on operations to update SQL tables by hand or to move data using ad hoc techniques is courting disaster. Mistakes get made in the heat of battle. Anticipate the corrective actions the operations team will need to make, and write and test these procedures up-front. Generally, the development team needs to automate emergency recovery actions and they must test them. Clearly not all failures can be anticipated, but typically a small set of recovery actions can be used to recover from broad classes of failures. Essentially, build and test "recovery kernels" that can be used and combined in different ways depending upon the scope and the nature of the disaster.

The recovery scripts need to be tested in production. The general rule is that nothing works if it isn't tested frequently so don't implement anything the team doesn't have the courage to use. If testing in production is too risky, the script isn't ready or safe for use in an emergency. The key point here is that disasters happen and it's amazing how frequently a small disaster becomes a big disaster as a consequence of a recovery step that doesn't work as expected. Anticipate these events and engineer automated actions to get the service back on line without further loss of data or up time.

- **Make the development team responsible.** Amazon is perhaps the most aggressive down this path

with their slogan “you built it, you manage it.” That position is perhaps slightly stronger than the one we would take, but it’s clearly the right general direction. If development is frequently called in the middle of the night, automation is the likely outcome. If operations is frequently called, the usual reaction is to grow the operations team.

- **Soft delete only.** Never delete anything. Just mark it deleted. When new data comes in, record the requests on the way. Keep a rolling two week (or more) history of all changes to help recover from software or administrative errors. If someone makes a mistake and forgets the where clause on a delete statement (it has happened before and it will again), all logical copies of the data are deleted. Neither RAID nor mirroring can protect against this form of error. The ability to recover the data can make the difference between a highly embarrassing issue or a minor, barely noticeable glitch. For those systems already doing off-line backups, this additional record of data coming into the service only needs to be since the last backup. But, being cautious, we recommend going farther back anyway.
- **Track resource allocation.** Understand the costs of additional load for capacity planning. Every service needs to develop some metrics of use such as concurrent users online, user requests per second, or something else appropriate. Whatever the metric, there must be a direct and known correlation between this measure of load and the hardware resources needed. The estimated load number should be fed by the sales and marketing teams and used by the operations team in capacity planning. Different services will have different change velocities and require different ordering cycles. We’ve worked on services where we updated the marketing forecasts every 90 days, and updated the capacity plan and ordered equipment every 30 days.
- **Make one change at a time.** When in trouble, only apply one change to the environment at a time. This may seem obvious, but we’ve seen many occasions when multiple changes meant cause and effect could not be correlated.
- **Make everything configurable.** Anything that has any chance of needing to be changed in production should be made configurable and tunable in production without a code change. Even if there is no good reason why a value will need to change in production, make it changeable as long as it is easy to do. These knobs shouldn’t be changed at will in production, and the system should be thoroughly tested using the configuration that is planned for production. But when a production problem arises, it is always easier, safer, and much faster to make a simple configuration change compared to coding, compiling, testing, and deploying code changes.

Auditing, Monitoring and Alerting

The operations team can’t instrument a service in deployment. Make substantial effort during development to ensure that performance data, health data, throughput data, etc. are all produced by every component in the system.

Any time there is a configuration change, the exact change, who did it, and when it was done needs to be logged in the audit log. When production problems begin, the first question to answer is what changes have been made recently. Without a configuration audit trail, the answer is always “nothing” has changed and it’s almost always the case that what was forgotten was the change that led to the question.

Alerting is an art. There is a tendency to alert on any event that the developer expects they might find interesting and so version-one services often produce reams of useless alerts which never get looked at. To be effective, each alert has to represent a problem. Otherwise, the operations team will learn to ignore them. We don’t know of any magic to get alerting correct other than to interactively tune what conditions drive alerts to ensure that all critical events are alerted and there are not alerts when nothing needs to be done. To get alerting levels correct, two metrics can help and are worth tracking:

1. alerts-to-trouble ticket ratio (with a goal of near one), and
2. number of systems health issues without corresponding alerts (with a goal of near zero).

Best practices include:

- **Instrument everything.** Measure every customer interaction or transaction that flows through the system and report anomalies. There is a place for “runners” (synthetic workloads that simulate user interactions with a service in production) but they aren’t close to sufficient. Using runners alone, we’ve seen it take days to even notice a serious problem, since the standard runner workload was continuing to be processed well, and then days more to know why.
- **Data is the most valuable asset.** If the normal operating behavior isn’t well-understood, it’s hard to respond to what isn’t. Lots of data on what is happening in the system needs to be gathered to know it really is working well. Many services have gone through catastrophic failures and only learned of the failure when the phones started ringing.
- **Have a customer view of service.** Perform end-to-end testing. Runners are not enough, but they are needed to ensure the service is fully working. Make sure complex and important paths such as logging in a new user are tested by the runners. Avoid false positives. If a runner failure isn’t considered important, change the test to one that is. Again, once people become accustomed to ignoring data, breakages won’t get immediate attention.

- **Instrument for production testing.** In order to safely test in production, complete monitoring and alerting is needed. If a component is failing, it needs to be detected quickly.
- **Latencies are the toughest problem.** Examples are slow I/O and not quite failing but processing slowly. These are hard to find, so instrument carefully to ensure they are detected.
- **Have sufficient production data.** In order to find problems, data has to be available. Build fine-grained monitoring in early or it becomes expensive to retrofit later. The most important data that we've relied upon includes:
 - **Use performance counters for all operations.** Record the latency of operations and number of operations per second at the least. The waxing and waning of these values is a huge red flag.
 - **Audit all operations.** Every time somebody does something, especially something significant, log it. This serves two purposes: first, the logs can be mined to find out what sort of things users are doing (in our case, the kind of queries they are doing) and second, it helps in debugging a problem once it is found.
A related point: this won't do much good if everyone is using the same account to administer the systems. A very bad idea but not all that rare.
 - **Track all fault tolerance mechanisms.** Fault tolerance mechanisms hide failures. Track every time a retry happens, or a piece of data is copied from one place to another, or a machine is rebooted or a service restarted. Know when fault tolerance is hiding little failures so they can be tracked down before they become big failures. We had a 2000-machine service fall slowly to only 400 available over the period of a few days without it being noticed initially.
 - **Track operations against important entities.** Make an "audit log" of everything significant that has happened to a particular entity, be it a document or chunk of documents. When running data analysis, it's common to find anomalies in the data. Know where the data came from and what processing it's been through. This is particularly difficult to add later in the project.
 - **Asserts.** Use asserts freely and throughout the product. Collect the resulting logs or crash dumps and investigate them. For systems that run different services in the same process boundary and can't use asserts, write trace records. Whatever the implementation, be able to flag problems and mine frequency of different problems.
- **Keep historical data.** Historical performance and log data is necessary for trending and problem diagnosis.
- **Configurable logging.** Support configurable logging that can optionally be turned on or off as needed to debug issues. Having to deploy new builds with extra monitoring during a failure is very dangerous.
- **Expose health information for monitoring.** Think about ways to externally monitor the health of the service and make it easy to monitor it in production.
- **Make all reported errors actionable.** Problems will happen. Things will break. If an unrecoverable error in code is detected and logged or reported as an error, the error message should indicate possible causes for the error and suggest ways to correct it. Un-actionable error reports are not useful and, over time, they get ignored and real failures will be missed.
- **Enable quick diagnosis of production problems.**
 - **Give enough information to diagnose.** When problems are flagged, give enough information that a person can diagnose it. Otherwise the barrier to entry will be too high and the flags will be ignored. For example, don't just say "10 queries returned no results." Add "and here is the list, and the times they happened."
 - **Chain of evidence.** Make sure that from beginning to end there is a path for developer to diagnose a problem. This is typically done with logs.
 - **Debugging in production.** We prefer a model where the systems are almost never touched by anyone including operations and that debugging is done by snapping the image, dumping the memory, and shipping it out of production. When production debugging is the only option, developers are the best choice. Ensure they are well trained in what is allowed on production servers. Our experience has been that the less frequently systems are touched in production, the happier customers generally are. So we recommend working very hard on not having to touch live systems still in production.
 - **Record all significant actions.** Every time the system does something important, particularly on a network request or modification of data, log what happened. This includes both when a user sends a command and what the system internally does. Having this record helps immensely in debugging problems. Even more importantly, mining tools can be built that find out useful aggregates, such as, what kind of queries are users doing (i.e., which words, how many words, etc.)

Graceful Degradation and Admission Control

There will be times when DOS attacks or some change in usage patterns causes a sudden workload spike. The service needs to be able to degrade gracefully and control admissions. For example, during 9/11 most news services melted down and couldn't provide a usable service to any of the user base. Reliably delivering a subset of the articles would have been a better choice. Two best practices, a "big red switch" and admission control, need to be tailored to each service. But both are powerful and necessary.

- **Support a "big red switch."** The idea of the "big red switch" originally came from Windows Live Search and it has a lot of power. We've generalized it somewhat in that more transactional services differ from Search in significant ways. But the idea is very powerful and applicable anywhere. Generally, a "big red switch" is a designed and tested action that can be taken when the service is no longer able to meet its SLA, or when that is imminent. Arguably referring to graceful degradation as a "big red switch" is a slightly confusing nomenclature but what is meant is the ability to shed non-critical load in an emergency.

The concept of a big red switch is to keep the vital processing progressing while shedding or delaying some non-critical workload. By design, this should never happen, but it's good to have recourse when it does. Trying to figure these out when the service is on fire is risky. If there is some load that can be queued and processed later, it's a candidate for a big red switch. If it's possible to continue to operate the transaction system while disabling advance querying, that's also a good candidate. The key thing is determining what is minimally required if the system is in trouble, and implementing and testing the option to shut off the non-essential services when that happens. Note that a correct big red switch is reversible. Resetting the switch should be tested to ensure that the full service returns to operation, including all batch jobs and other previously halted non-critical work.

- **Control admission.** The second important concept is admission control. If the current load cannot be processed on the system, bringing more work load into the system just assures that a larger cross section of the user base is going to get a bad experience. How this gets done is dependent on the system and some can do this more easily than others. As an example, the last service we led processed email. If the system was over-capacity and starting to queue, we were better off not accepting more mail into the system and let it queue at the source. The key reason this made sense, and actually decreased overall service latency, is that as our queues built, we

processed more slowly. If we didn't allow the queues to build, throughput would be higher. Another technique is to service premium customers ahead of non-premium customers, or known users ahead of guests, or guests ahead of users if "try and buy" is part of the business model.

- **Meter admission.** Another incredibly important concept is a modification of the admission control point made above. If the system fails and goes down, be able to bring it back up slowly ensuring that all is well. It must be possible to let just one user in, then let in 10 users/second, and slowly ramp up. It's vital that each service have a fine-grained knob to slowly ramp up usage when coming back on line or recovering from a catastrophic failure. This capability is rarely included in the first release of any service.

Where a service has clients, there must be a means for the service to inform the client that it's down and when it might be up. This allows the client to continue to operate on local data if applicable, and getting the client to back-off and not pound the service can make it easier to get the service back on line. This also gives an opportunity for the service owners to communicate directly with the user (see below) and control their expectations. Another client-side trick that can be used to prevent them all synchronously hammering the server is to introduce intentional jitter and per-entity automatic backup.

Customer and Press Communication Plan

Systems fail, and there will be times when latency or other issues must be communicated to customers. Communications should be made available through multiple channels in an opt-in basis: RSS, web, instant messages, email, etc. For those services with clients, the ability for the service to communicate with the user through the client can be very useful. The client can be asked to back off until some specific time or for some duration. The client can be asked to run in disconnected, cached mode if supported. The client can show the user the system status and when full functionality is expected to be available again.

Even without a client, if users interact with the system via web pages for example, the system state can still be communicated to them. If users understand what is happening and have a reasonable expectation of when the service will be restored, satisfaction is much higher. There is a natural tendency for service owners to want to hide system issues but, over time, we've become convinced that making information on the state of the service available to the customer base almost always improves customer satisfaction. Even in no-charge systems, if people know what is happening and when it'll be back, they appear less likely to abandon the service.

Certain types of events will bring press coverage. The service will be much better represented if these scenarios are prepared for in advance. Issues like mass data loss or corruption, security breach, privacy violations, and lengthy service down-times can draw the press. Have a communications plan in place. Know who to call when and how to direct calls. The skeleton of the communications plan should already be drawn up. Each type of disaster should have a plan in place on who to call, when to call them, and how to handle communications.

Customer Self-Provisioning and Self-Help

Customer self-provisioning substantially reduces costs and also increases customer satisfaction. If a customer can go to the web, enter the needed data and just start using the service, they are happier than if they had to waste time in a call processing queue. We've always felt that the major cell phone carriers miss an opportunity to both save and improve customer satisfaction by not allowing self-service for those that don't want to call the customer support group.

Conclusion

Reducing operations costs and improving service reliability for a high scale internet service starts with writing the service to be operations-friendly. In this document we define operations-friendly and summarize best practices in service design, development, deployment, and operation from engineers working on high-scale services.

Acknowledgments

We would like to thank Andrew Cencini (Rackable Systems), Tony Chen (Xbox Live), Filo D'Souza (Exchange Hosted Services & SQL Server), Jawaid Ekram (Exchange Hosted Services & Live Meeting), Matt Gambardella (Rackable Systems), Eliot Gillum (Windows Live Hotmail), Bill Hoffman (Windows Live Storage Platform), John Keiser (Windows Live Search), Anastasios Kasiolas (Windows Live Storage), David Nichols (Windows Live Messenger & Silverlight), Deepak Patil (Windows Live Operations), Todd Roman (Exchange Hosted Services), Achint Srivastava (Windows Live Search), Phil Smoot (Windows Live Hotmail), Yan Leshinsky (Windows Live Search), Mike Ziock (Exchange Hosted Services & Live Meeting), Jim Gray (Microsoft Research), and David Treadwell (Windows Live Platform Services) for background information, points from their experience, and comments on early drafts of this paper. We particularly appreciated the input from Bill Hoffman of the Windows Live Storage team and Achint Srivastava and John Keiser, both of the Windows Live Search team.

Author Biography

James Hamilton is an architect on the Microsoft Live Platform Services team and has been with Microsoft for just over ten years. Previously, he led the

Exchange Hosted Services team that provided email-related services to over two million users. He spent much of his first eight years at Microsoft as a member of the SQL Server team, where he led most of the core engine development teams.

Before joining Microsoft, James was lead architect for IBM's DB2 UDB relational database system, and earlier led the delivery of IBM's first C++ compiler. In the late 70's and early 80's he worked as a licensed auto mechanic servicing and racing exotic Italian cars. James' web site is <http://research.microsoft.com/~jamesrh> and his email is JamesRH@microsoft.com.

References

- [1] Isard, Michael, "Autopilot: Automatic Data Center Operation," *Operating Systems Review*, April, 2007, <http://research.microsoft.com/users/misard/papers/osr2007.pdf>.
- [2] Patterson, David, *Recovery Oriented Computing*, Berkeley, CA, 2005, <http://roc.cs.berkeley.edu/>.
- [3] Patterson, David, *Recovery Oriented Computing: A New Research Agenda for a New Century*, February, 2002, <http://www.cs.berkeley.edu/~pattsrn/talks/HPCAkeynote.ppt>.
- [4] Fox, Armando and D. Patterson, "Self-Repairing Computers," *Scientific American*, June, 2003, <http://www.sciam.com/article.cfm?articleID=000DAA41-3B4E-1EB7-BDC0809EC588EEDF>.
- [5] Fox, Armando, *Crash-Only Software*, Stanford, CA, 2004, <http://crash.stanford.edu/>.
- [6] Hoffman, Bill, *Windows Live Storage Platform*, private communication, 2006.
- [7] Shakib, Darren, *Windows Live Search*, private communication, 2004.
- [8] *Writing Secure Code, Second Edition*, Howard, Michael, and David C. LeBlanc, <http://www.amazon.com/Writing-Secure-Second-Michael-Howard/dp/0735617228>.

RepuScore: Collaborative Reputation Management Framework for Email Infrastructure

Gautam Singaraju and Brent ByungHoon Kang – University of North Carolina at Charlotte

ABSTRACT

We propose RepuScore, a collaborative reputation management framework over email infrastructure, which allows participating organizations to establish sender accountability on the basis of senders' past actions. RepuScore's generalized design can be deployed with any Sender Authentication technique such as SPF, SenderID and DKIM. With RepuScore, participating organizations collect information on sender reputation locally from users or existing spam classification mechanisms and submit it to a central RepuScore authority. The central authority generates a global reputation summary which can be used to enforce sender accountability. We present the algorithms for reputation score calculation and share our findings from experiments based on a RepuScore prototype using a) our simulation logs and b) a 20 day log from a non-profit organization with five collaborating domains.

Introduction

In an effort to prevent sender address spoofing and phishing attacks, about 35% of all emails over the Internet are authenticated using sender authentication systems [13] such as DKIM [1, 23], SPF [22] and SenderID [11]. These systems allow receivers to authenticate the sender's mail server before email delivery to a mailbox.

Authentication schemes alone, however, do not provide the organization with the capability to differentiate between a credible sender and an unscrupulous one. Indeed, it has been noted that spammers have been the early adopters of these systems. This shows that a sender's identity does not necessarily guarantee their trustworthiness because trusting a sender can only be possible after verifying their past adherence to best mail practices. Currently, sender identification techniques are being used as the basis for determining the sender's history of adherence to best mail practices [21]. A reputation management system deployed at a single organization [3] has demonstrated that the history of the sender's adherence can provide an effective email classification mechanism.

We believe that organizations would benefit from sharing senders' reputation information that is individually collected at each domain. By collecting reputation scores from multiple organizations, the email receivers could access a complete history of a sender's past actions. Such a global perspective of a sender's reputation would allow receivers to trust a sender that they have no prior information about. As with receiver collaboration, a sender's spamming activity would be reported to all receivers: the onus is on the senders not to transmit unsolicited emails to any reputation-sharing receiver.

In this paper, we propose RepuScore, a reputation management framework for email infrastructure that uses receiver collaboration to compile global reputation for a sender. RepuScore helps create and maintain a trusted group of organizations. We discuss the deployment of RepuScore with sender authentication techniques. The design considerations for RepuScore are as follows:

First, the RepuScore framework can be used to collect, compute and share reputation among organizations. To keep track of the sender's history of adherence, RepuScore takes into account the reputation of the sender in the previous time frame along with the spam rate in the present time frame. Towards this, RepuScore employs the Time Sliding Window Exponentially Weighted Moving Average (TSW-EWMA) algorithm.

Second, RepuScore eases the overhead of reputation collection and computation with the help of a distributed architecture. Such architecture allows each organization to collect votes from its users. However, distributing the reputation management creates additional challenges.

Since RepuScore employs a distributed reputation framework, it is susceptible to Sybil attacks [20, 26]. In Sybil attacks, a malicious receiver manipulates the rating mechanism by creating multiple identities to give a higher rating to emails sent from the colluding senders and a lower rating to legitimate senders. Sybil attacks are thwarted by valuing a reputable participant's rating more highly than that of a less reputable participant. RepuScore employs the Weighted Moving Algorithm Continuous (WMC) [24] to thwart Sybil attacks. RepuScore introduces a participant voting threshold, a minimum threshold required by organizations to

participate in global reputation computation, to mitigate Sybil attacks.

Third, RepuScore supports a centralized reputation scoring mechanism with minimal overhead. This centralized mechanism creates a trusted group of reputable senders. The lack of centralized enforcement has been cited as the main obstacle in tying email fraud to a particular user or organization [10].

The remainder of this paper is organized as follows. In the following section, we discuss the related work. We then describe the design issues for a reputation management framework and present the RepuScore design in the next section followed by its prototype implementation. We then discuss our results and conclude in the last sections.

Related Work

In this section, we discuss the reputation management frameworks that have been designed for email infrastructure, followed by a discussion on sender identity systems.

Reputation Systems for Email Infrastructure

SenderPath's Sender Score [19] and Habeas' SenderIndex [8] provide reputation for a sender's IP address. SecureComputing's TrustedSource [4] provides a global reputation system with the help of deployed mail servers in different organizations. Reputation based on IP addresses is not effective, as an IP address cannot be bonded to a specific organization [5]. For instance, when multiple organizations share an IP address, spammers in a single domain can affect the reputation of users in other organizations. Moreover, if organizations move to another service provider, their past actions would no longer be attributed to them. We believe that a reputation should be more closely associated with the organization, possibly utilizing the domain name of the organization.

Project Lumos [9] was proposed as an effort to provide reputation among collaborating ISPs. The receivers provided feedback as to whether a sender was a spammer or otherwise. Reputation was based on the activity of the previous 180 days. Project Lumos was designed to consider the weighted average of previous and present reputation of the senders. We believe that to thwart Sybil attacks and provide an open reputation management system, the reporter's reputation should also be taken into account in order to provide an accurate summary of a sender's reputation.

Google's reputation service [3] identifies the senders using best-guess SPF [22] or DKIM [1, 23] and computes the sender's reputation based on the inputs from users. This system demonstrated a high accuracy in classifying Google's emails. The paper also points out the need for a third party reputation framework.

Certification Systems for Email Infrastructure

Systems like SenderPath's SenderScore Certified [18], Habeas' Safelist [7] and Goodmail's Certified

Email [6] are certification and accreditation services. These services allow bulk senders to obtain third party certification to be able to send bulk emails. They are not really reputation systems, as the sender maintains the reputation and not the receivers.

Identity Based Email Classification

Receivers can identify senders based on the sender's email ID, IP address or domain. For instance, PGP [15, 27] is an email Id-based authentication technique where a third-party server maintains individual users' public keys. The receivers verify the senders' signed emails by retrieving the sender's public key.

IP addresses are used to identity spammers in systems such as Blacklist IP and Real-time Blackhole List (RBL) [17] that keep a list of IP addresses that propagate spam. Though several RBLs are available, a recent study has shown that only 50% of spam is correctly identified by combined use of two or more lists [16].

We believe that maintaining a group of high-spam-propagating domains is more difficult than maintaining a group of non-spam-propagating domains. Spamming senders usually do not exist for long periods of time, whereas non-spamming senders usually exist for long periods of time.

SenderID [11] verifies the IP address presented by the email against that of the sender's registered mail servers. Using Sender Policy Framework (SPF) [22], a receiver thwarts sender forgery by identifying the sender's mail server through DNS entries. Domain Key Identified Mail (DKIM) [1, 13] publishes the mail server's public keys as a part of DNS records. Each email is signed by the sender's mail server. The signature is used by the receiver's mail server to verify the sender.

Accredited DomainKeys adds a central authority to DomainKeys architecture [12]. The centralized authority, called the Accreditation Bureau, maintains the sender domain's public key. The users should conform to a specified usage policy and adherence to the policy is checked periodically. We suggest that a reputation based mechanism where the receivers can vote on whether the senders adhere to the specified usage policy would help in enforcement of the usage policy.

RepuScore Design

In this section, we describe the design considerations for a reputation framework. We note that authentication techniques and a reputation framework work together to create a trusted group of reputable senders. A verified identity (through an existing authentication mechanism) is a required basis for maintaining sender's reputation. Moreover, a reputation service is able to guide a receiver through the process of validating the sender before the sender's emails are accepted

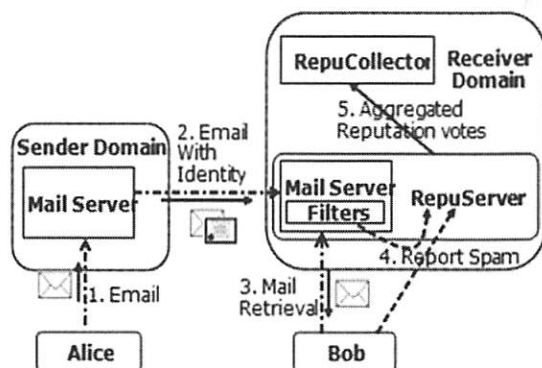
Sender Identity Techniques

Email Id-identity systems, such as PGP, can be used to maintain reputation. However, using email ids

entails maintaining a huge overhead for vote collection, storage and reputation computation. Instead of using email ids as identities for reputation management, we use domain authentication schemes, thereby decreasing the number of identities needed. We believe that this approach is more scalable than the email id based reputation system.

As mentioned, about 35% of all authenticated email over the Internet is authenticated using SPF, DKIM or SenderID [13]. A reputation management system can be built to help evaluate the senders who are being authenticated using these mechanisms. Such a mechanism will help evaluate the domains that adhere to a common guideline.

The lack of a centralized authority [10] has been noted as a main reason for the inability to tie email forgery to a single user or the organization. A central authority can maintain a trusted group of reputable senders where each sender needs to maintain a high reputation. Such a mechanism allows a common best email practice to be enforced among senders.



Listing 1: Collection of reputation votes by different RepuServers in an organization. The sender's identity could be used with any domain authentication technique. The reputation votes can be submitted either by users such as Bob or using any currently available filtering mechanism. Each domain maintains a single RepuCollector that collects the reputation votes from the multiple RepuServers in the organization.

Design of RepuScore Framework

A reputation management framework should only accept a single reputation vote from each organization. A large global organization might have multiple mail servers, each situated in different geographic locations, for example, in different countries. If a reputation management framework considered votes from mail servers, an organization with a huge number of mail servers would have greater say than organizations with a single mail server. Hence, each organization should be given a single vote that should be the aggregate of all the mail servers in the domain.

We define RepuServer as a mail server with the capability of verifying the users and collecting the

reputation votes from them. Each local RepuServer at a domain collects votes from its users and email filters, aggregates the votes locally, and forwards them to the RepuCollector of the domain. We define a RepuCollector as an organizational level service that aggregates the votes from the local RepuServers and participates in a global reputation of peer RepuCollectors.

Each RepuServer records the total number of emails received during a given period and counts the ones that are considered to be spam by the currently available spam-filtering mechanism or by the user's input. Figure 1 demonstrates the mechanism for the receivers to report spam to their local RepuServer. Such reports can also be performed by currently available email classification techniques without user involvement. In the event that the report is conflicting, a user's input can be used to increase the reputation of the sender.

The RepuCollector's reputation should decrease for bad behavior and increase in the absence of bad behavior. For example, if spam is reported, the sender's reputation should decrease. If no spam is reported, the reputation should increase.

An ideal initial reputation is a requirement for building the reputation of a new RepuCollector or RepuServer. An improper initial reputation would give high spam propagating domains an unfair advantage as their reputation would stay high for a long time. In contrast, a low initial reputation would be unfair to a new domain as its emails would not be accepted by peers.

Since the sender's reputation changes over time and is computed after receiver collaboration, the reputation is computed in every time period. We define this period as the Reputation Aggregation Interval. A RepuCollector should invest a significant number of reputation aggregation intervals to be considered a good sender. Such a mechanism would make spamming unviable for a spammer as it would require a significant investment of resources, including both time and money. In addition, a quick reduction in reputation for non-adherence to the policy removes spammers from the trusted group of senders.

Finally, the reputation framework should guard against Sybil attacks [20, 26] where users with multiple identities attempt to change the reputation of the senders [24]. We believe that domains which transfer high amounts of spam would attempt to unfairly increase their reputations in order to be considered part of the trusted group of senders. To thwart such attacks, RepuCollectors having a reputation lower than a given threshold, which we refer to as the Participation Threshold, would not be allowed to participate in the voting mechanism.

RepuScore Prototype Implementation

Our framework, RepuScore, is a generic design that can be employed by sender identity systems. As

discussed in the related work section, RepuScore creates a trusted group of reputable senders. We believe that a reputation framework should facilitate creation of such a group rather than just maintaining a group of blacklisted senders. In this section, we describe a generalized architecture that can be used with RepuScore. We generalize the architecture so that any sender authentication scheme can be used along with RepuScore. The architecture describes vote collection, reputation computation and also centralized reputation computation. Finally, we discuss the RepuScore algorithm used for reputation computation.

RepuScore differs from other approaches because of the collaborative reputation based on the scores suggested by peers. As RepuScore is designed as a collaborative mechanism, it has been designed to protect against Sybil Attacks, where a single attacker can take multiple identities. Towards this, RepuScore takes into account the reputation of the reporting server along with the reputation they report for a peer.

RepuScore Architecture

RepuScore's hierarchical architecture is designed so that the reputation collection and computation is manageable as the number of participating domains increase. The RepuScore framework computes reputation based on the votes collected by each RepuServer. While collecting reputation votes, a RepuServer checks the validity of the reporting users. The user's votes are based on their evaluation of the sender's adherence to best practices. We outline three major steps in RepuScore's architecture:

- a) Reputation Vote Collection
- b) Reputation Computation
- c) Reputation Lookup Service

Reputation Vote Collection

As the definition of spam is subjective, an email regarded as spam by one user might not be considered so by another. Therefore, a global blacklist or white list would not be ideal as it would fail to represent the conflicting views of multiple users. RepuScore employs a social rating mechanism to consider the conflicting views of the users.

The receiver's RepuServer can maintain the number of emails received and the emails marked as spam for each sender RepuServer. The vote collection mechanism should require minimal participation from the users. For example, RepuServer collects the users' votes based on the users' implicit inputs. Users only need to mark an incorrectly filtered-email as non-spam or to report a spam email that was not correctly filtered by the spam classifiers. (Many email services provide similar mechanisms for their users to report a spam email or an incorrectly filtered email.) Figure 1 demonstrates the mechanism in which the RepuServer collects the votes from multiple users. Before accepting votes from the users, the RepuServer should validate the users.

The spam classifiers are also used along with users' input in collecting votes. A negative vote for a sender is entered when the spam filters determines an email as spam. Likewise, a positive vote for a sender is automatically made when the sender's email is not considered spam. In the event that the spam filter marks a legitimate email as spam, the users can mark the email as non-spam, submitting a positive vote for a sender to the RepuServer.

Reputation Computation

Based on the number of spam and emails collected, each RepuServer calculates the reputation of the sender RepuServer. RepuServer Reputation is defined as the weighted average of its reputation in the previous reputation aggregation interval and the reputation computed in the present reputation aggregation interval.

RepuScore calculates the reputation of a RepuCollector based on the reputation of the RepuServers maintained by the RepuCollector. We define the RepuCollector Reputation as the aggregate reputation of the RepuServers in their domain in the present reputation aggregation interval.

Each RepuCollector calculates the local reputation for each peer RepuCollector. The computed reputation is digitally signed by each RepuCollector to maintain the integrity of the data. To provide a global perspective, the locally computed RepuCollector's reputations should be collected by the Central Authority.

RepuScore introduces a central authority that collects reputation votes from all the RepuCollectors and computes the global reputation for all RepuCollectors. The central authority verifies the RepuCollector's votes based on the digital signature. The central authority should make sure that the reputation collection is conducted once every reputation aggregation interval. The central authority calculates a global reputation for each RepuCollector based on the change in its reputation as reported by peer RepuCollectors. The central authority takes into account the reputation of the RepuCollectors to compute the global reputation of the peer RepuCollectors. If the reporting RepuServers' reputation is below the participation threshold, their reputation votes are not factored into the global reputation.

Reputation Lookup Service

A reputation Lookup service can be provided with the help of a third party lookup service. The reputation lookup service can be similar to Realtime Black Lists. Such a reputation look up service can also provide a mechanism for the receivers to lookup the reputation of a sender's RepuCollector as reported by peers.

An alternate way for receivers to determine reputation is by associating the reputation with a sender identity that can be vouched for by a third party. For example, in the case of Accredited DomainKeys, the reputation can be embedded as the part of the seal that is

supplied to the MTAs. When the client checks the DNS entries, the seal can be verified for the reputation.

RepuScore Algorithm

In this section, we discuss RepuScore's algorithm. The RepuCollector's reputation is calculated based on the reputation of all the RepuServers it maintains. We first demonstrate how each RepuCollector calculates the reputation of peer RepuServers. We then discuss the reputation computation of peer RepuCollectors. Finally, we demonstrate how a global reputation is calculated.

With the help of reputation, administrators in an organization can evaluate the compliance of the domain by checking their organization's reputation services. If the domain's reputation is lower than expected that would imply that there might be bots on the server [14].

RepuServer Reputation Calculation

As discussed in the above sections, a RepuServer's reputation is calculated by peer RepuServers. The reputation in RepuScore is always in the open interval (0, 1). A score of 1 indicates a highly reputable sender whereas a score of 0 indicates a sender with a low reputation. For all sender RepuServers, each receiving RepuServer maintains the number of emails received and the number among those marked as spam. The reputation of a RepuServer is computed as the number of good emails over the number of emails sent by a RepuServer in a particular interval. The reputation is calculated based on the modified time sliding window exponentially weighted moving average (TSW-EWMA) algorithm [2].

Equation 1 displays the weighted moving average. The RepuServer Reputation is based on the reputation in the previous interval and the reputation in the present interval. Correlation factor α indicates the amount of previous reputation considered for computation of the RepuServer's reputation in the new interval. If the correlation factor is high, the reputation of a sender takes a long time to increase or decrease, as a lot of weight is given to the previous reputation. However, if the correlation factor is low, the reputation increases or decreases very quickly since current actions are given additional weight. We demonstrate the effect of the correlation factor on reputation in our experiments.

RepuCollector Reputation Calculation

Based on the change in a RepuServers' reputation, the RepuCollector's reputation can be updated. Equation 2 shows the local reputation computation of a RepuCollector. The local RepuCollector reputation is the average reputation of all of its RepuServers. Each RepuServer transmits the reputation of the peer RepuCollectors to the central authority. As discussed in the above sections, the central authority considers the votes only from the RepuCollectors whose reputation is greater than the participation threshold. We

Given:

- E_m is the set of emails received by RepuServer in reputation aggregation interval m
- S_i is the RepuServer i

For all $RS_i \in \text{Set } E_m$:

- $\text{NewRep}(S_i, m) = 1 - \frac{n(\text{spam}_m)}{n(\text{TotalEmails}_m)}$
- $\text{Rep}(S_i, m) = \alpha \times \text{Rep}(S_i, m-1) + (1 - \alpha) \times \text{NewRep}(S_i, m)$

Where:

- $\text{Rep}(S_i, m)$ is Sender RepuServer's Reputation in the interval m .
- $n(\text{spam}_m)$ is the number of spam received in the interval m .
- $n(\text{TotalEmails}_m)$ is the total number of emails received in the interval m .
- α ($0 \leq \alpha \leq 1$) is the correlation factor between the previous and the present value.

Equation 1: Local RepuServer reputation.

Given:

- RepuCollector reputation is the average of the reputation reported to a RepuCollector's by its RepuServers.

For all RepuCollectors:

$$\text{LocalCollectorRep}_{co}(m) = \frac{1}{n(S_p)} \sum_{i=0}^{n(RW)} \text{repu}(S_i, m)$$

Where:

- $\text{LocalCollectorRep}_{co}$ is the local reputation of RepuCollector c reported by RepuServer o in the organization.
- $n(S_p)$ is the number of Sender RepuServer seen by a RepuServer o .

Equation 2: Local RepuCollector reputation.

Given:

- The RepuCollector reputation is weighted moving average continuous of local reputation computed in the m th Interval.

For all RepuCollectors, Central Authority calculates:

$\text{CollectorRep}_i(m) =$

$$\sum_{n=0}^i \frac{\text{CollectorRep}_n(m-1) \times \text{LocalCollectorRep}_n(m)}{\sum_{n=0}^i \text{CollectorRep}_n(m-1)}$$

Where:

- $RD\text{-Repu}_i$ is the global reputation of RepuCollector i .

Equation 3: Global RepuCollector reputation.

demonstrate that such a mechanism helps in the creation of a trusted group of reputable senders.

The central authority calculates the global reputation of the RepuCollectors based on a modified Weighted Majority Algorithm (WMA) called WMA Continuous (WMC) proposed by Yu et al [24]. The WMC algorithm has been used in peer-to-peer systems to detect deception. We provide the participation threshold as a mechanism to remove domains that

propagate spam and increase reputations of other senders.

Equation 3 demonstrates the Global RepuCollector reputation as the reputation-weighted average of the local RepuCollector reputation computed by each peer. The new reputation is computed once every reputation aggregation interval and is valid for one Aggregation Interval.

Experiments and Results

In this section, we demonstrate the effectiveness of RepuScore through experiments. We accomplish this with the help of a) simulated logs to demonstrate specific properties of RepuScore and b) real logs from a non-profit organization. The logs from the organization were 20-day logs collected by five domains that they maintained. The log contained information about 45K domains to which about 450K emails were sent, 55% of which were marked as spam by RBLs or rejected since the sender domain were determined not to exist through DNS reverse lookup.

Effect of α on Reputation of a Trusted RepuCollector With Sudden Increase in the Amount of Spam it Transmits

Spammers might attempt to thwart RepuScore by building reputation and then suddenly transmitting huge amounts of spam. In such cases, it is expected that the reputation of the sender would decrease and the spammer would be removed from the trusted group within a minimal number of reputation aggregation intervals.

To demonstrate the effectiveness of RepuScore, we created logs with 100 RepuCollectors spanning 45 reputation aggregation intervals. We selected a random

number of RepuServers which reported to their local RepuCollectors. The number of emails and spams that were transmitted to and from an organization was perturbed using a random number; for example, since RepuScore creates a trusted group of reputable senders, the spam rate among them was set at under 20%, whereas a spamming domain's spam rate was set at greater than 95%. (We see this trend in the logs from the non-profit organization.)

Figure 2 demonstrates the reputation of a RepuCollector from which the amount of spam suddenly increased as a function of α . For the first 30 reputation intervals, the RepuCollector built its reputation and attempted to be a part of the trusted group. After reputation interval 30, the spam rate from the RepuCollector increased to 95%. The RepuCollector's reputation is based on the reputation of all its RepuServers. The jump in the value of reputation is due to the value of α and the initial reputation value of RepuDomain that was set at 0.5. Therefore, the reputation of the RepuCollector for $\alpha = 0.9$ decreased from 0.7 after the first reputation aggregation interval. In cases where the sender does not propagate spam, the reputation should increase slowly, which indicates a long past history. Hence the high value of α implies an association for a long history of good actions. If the sender propagates spam, the reputation should decrease immediately, reflecting the current actions of the sender. A low value of α guarantees an immediate reduction when the sender propagates spam. Equation 4 demonstrates our change in the reputation algorithm to accommodate this behavior. Figure 3 demonstrates the change in reputation by employing the modified algorithm. For a high α , the reputation increases gradually but decreases more rapidly.

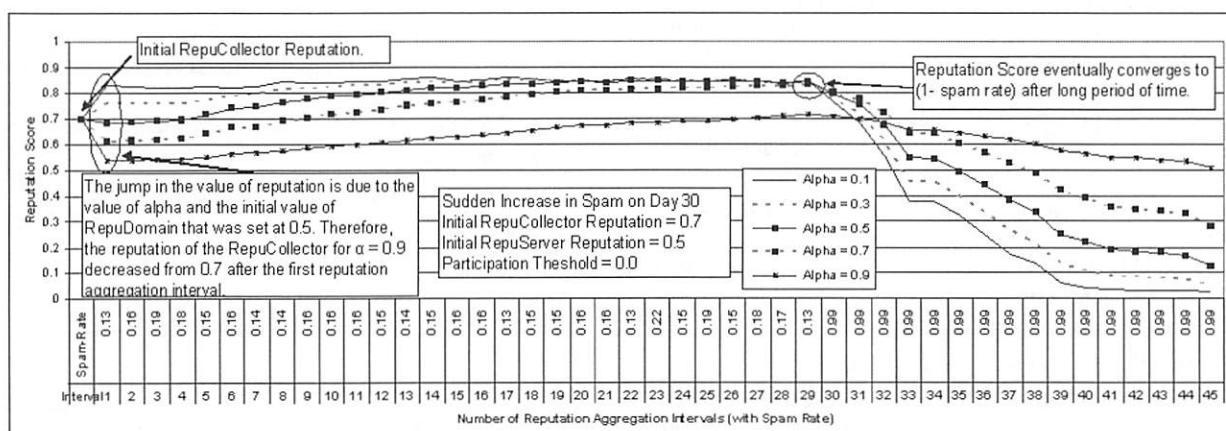


Figure 2: demonstrates the change in the reputation score of a trusted domain that transmits spam after reputation interval 30 as a function of α . The reputation eventually converges to $(1 - \text{average spam rate})$ over multiple reputation intervals. High α puts more weight to previous reputation score, whereas low α puts more weights to current score. Thus, for high values of α , it takes long time for the reputation to be built up whereas for low α value the decrease (or increase) in reputation is faster. The sudden drop from the initial score to the first interval is due to the effect of α . The RepuCollector's reputation has been set at 0.7. In the future intervals, the RepuCollector reputation is based on the reputation of all RepuServers which starts at 0.5. Therefore, for $\alpha = 0.9$, the reputation of RepuDomain is around 0.55.

$$\text{If } (\text{Rep}(S_i, m-1) \geq \text{NewRep}(S_i, m)) \\ \text{Rep}(S_i, m) = \alpha \times \text{Rep}(S_i, m-1) + \\ (1 - \alpha) \times \text{NewRep}(S_i, m)$$

Else

$$\text{Rep}(RS_i, m) = (1 - \alpha) \times \text{Rep}(RS_i, m-1) + \\ \alpha \times \text{NewRep}(S_i, m)$$

Equation 4: Local ReputServer reputation.

Figure 4 shows the modified ReputScore algorithm with collaboration among multiple domains using the 20 day logs from the non-profit organization. The reputation of the spamming domain decreased, but the reputation of a good domain increased.

Participation Threshold and Initial Values for ReputCollector

Having an appropriate initial value for ReputCollector's reputation is extremely important to maintain a trusted group of reputable senders. For instance, if the initial reputation scores for the ReputCollector and ReputServers are set too high, it would take a long time

for the reputation to decrease. On the other hand, if the initial reputation is set too low, it would take a long time for the reputation of a non-spamming ReputCollector to increase.

Our experiments show that an ideal initial reputation value for the ReputServer and the ReputCollector is between 0.5 and 0.7. With different initial values we noted that the average reputation of all the domains using the logs from the non-profit organization converged to about 0.6 for $\alpha = 0.1$, 0.47 for $\alpha = 0.5$ and 0.36 for $\alpha = 0.9$. Hence, an ideal initial reputation should be equal to the average reputation of all domains in the system after a long period of time. In order for the new reputation domains to participate in the reputation aggregation intervals, the threshold should be 0.1-0.3 below the initial reputation.

Resilience to Sybil Attacks

We increased the percentage of malicious ReputCollectors from 10 to 30% to demonstrate ReputScore's

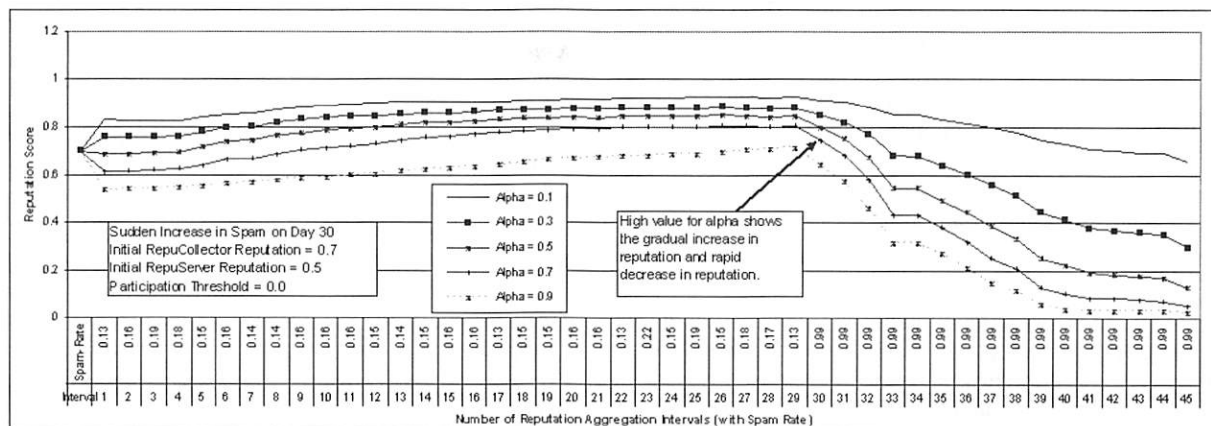


Figure 3: In the modified ReputScore algorithm, a high value of α (other than 1.0) implies gradual increase, but fast decrease in reputation when the domain starts to spamming.

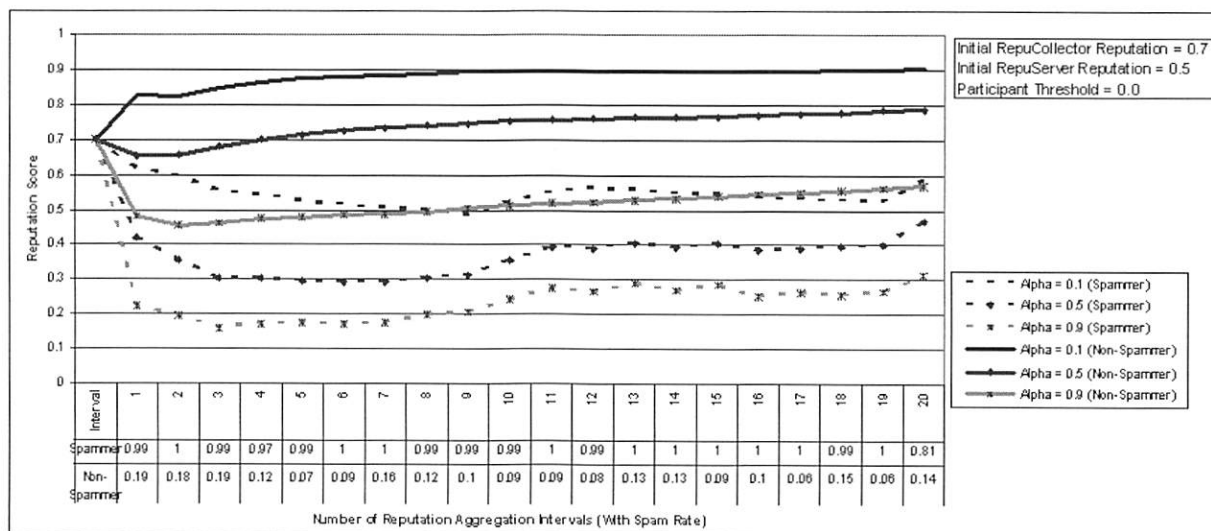


Figure 4: Using modified ReputScore algorithm with 20 days log from a non-profit organization. A number of new ReputCollectors were introduced at different reputation aggregation intervals.

resilience to Sybil attacks. Each RepuCollector transmits a high amount of spam ($> 95\%$) for the first 30 reputation aggregation intervals. After 30 reputation intervals, we had the Sybil attacker to start increasing the reputation of its own Sybil domains and decrease the reputation of other domains. Figure 5 demonstrates our results. The reputation of the Sybil domains steadily decreased, but the reputation of the non-Sybil domains increased.

Conclusion

RepuScore is a collaborative reputation framework that collects votes from multiple organizations in order to collectively compute the reputation of a sender. We believe that RepuScore is a step toward enforcing sender accountability through collaboration among domains.

Simply blacklisting spammers is ineffective because spammers continue to easily create new sender identities. In contrast, a legitimate sender's identity typically exists for long periods of time. Thus, we believe a reputation framework such as RepuScore will be more effective in blocking spam email by maintaining a group of reputable trusted senders rather than identifying spamming domains.

RepuScore distributes the overhead for reputation collection and computation by using a distributed architecture while allowing a centralized authority to collectively calculate the global reputation for each sender domain.

Our experiments using simulated logs and an actual log from a non-profit organization demonstrated RepuScore's effectiveness and its ability to thwart Sybil attacks. We also presented the algorithms for reputation score calculation and demonstrated the effect of the correlation factor α where a sender's reputation increases gradually when it does not propagate spam but decreases immediately when it transmits spam.

Availability

RepuScore will be an open-source effort aimed to provide participating domains with the ability to contribute information about senders and also lookup the collected reputation about them. RepuScore will be made available from <http://isr.uncc.edu/RepuScore>.

Acknowledgement

We would like to thank our shepherd, Peter Galvin, and the anonymous reviewers for their insightful comments. We would also like to show our appreciation to our LISA copy-editor, Rob Kolstad, for his excellent help. Finally, we thank our ISR lab member Sumeet Jain, for his help from the early stage of this paper.

Author Biographies

Gautam Singaraju is a fifth year doctoral student at University of North Carolina at Charlotte and is advised by Dr. Kang. Previously, he completed M.S in Computer Science at UNC Charlotte and a B.Tech in Electronics and Communication Engineering at JNTU, India. Since 2003, he has also been a volunteer System Administrator for a global non-profit organization. During the summer of 2007, he has worked at VMware with the performance group. Gautam can be reached at gsingara@uncc.edu.

Brent Hoon Kang received his Ph.D in Computer Science from the University of California at Berkeley, working on the Berkeley Digital Library and OceanStore project. Prior to Berkeley, he received an M.S in Computer Science from the University of Maryland at College Park, and a B.S in Computer Science and Statistics from Seoul National University. Since Fall 2004, he has been an assistant professor at the University of North Carolina (UNC) at Charlotte, and has been leading the Infrastructure Systems Research (ISR) Lab. As part of his research efforts, he recently worked on IT

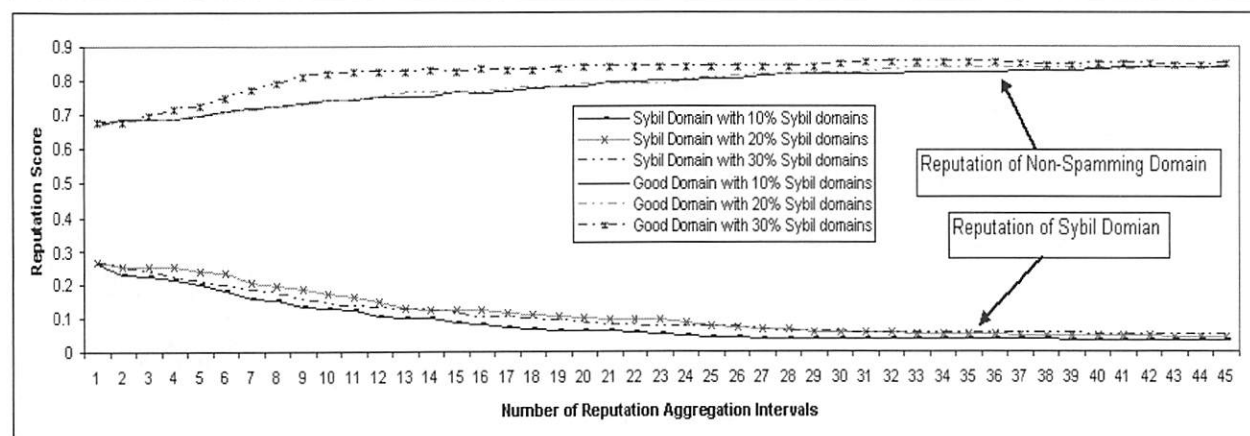


Figure 5: Multiple spamming domains (under a Sybil attacker's control) increase their votes for each other after the reputation aggregation interval 30. The domains give each other high reputation scores and attempt to decrease the reputation of other domains. However, our RepuScore framework was resilient to Sybil attack. The participation threshold was set to 0.

infrastructure design and administration issues related to protecting infrastructure against security threats such as the bots/malwares and the email spam/phishing problems. As part of his efforts on Information Assurance (IA) education program, he has been developing the hands-on cyber exercise components that foster students' creativeness and problem solving skills for IT systems design and defense. Hoon can be reached at bbkang@uncc.edu.

Bibliography

- [1] Allman, E., *DomainKeys Identified Mail (DKIM): Introduction and Overview*, 2005, <http://mipas-soc.org/dkim/info/DKIM-Intro-Allman.html>.
- [2] Biswas, S. and R. Morris, "ExOR: Opportunistic Multi-Hop Routing for Wireless Networks," *Proceedings of ACM SIGCOMM '05*, Philadelphia, 2005.
- [3] Taylor, Bradley, "Sender Reputation in a Large Webmail Service," *Third Conference on Email and Anti-Spam (CEAS 2006)*, 2006.
- [4] CipherTrust, *TrustedSource: The Next-Generation Reputation System*, White Paper, 2006.
- [5] Dewan, P. and Dasgupta, P., "Pride: Peer-to-Peer Reputation Infrastructure for Decentralized Environments," *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, pp. 480-481, ACM Press, New York, NY, USA, May 19-21, 2004.
- [6] Goodmail Systems, *Certified Email*, <http://www.goodmailsystems.com/certifiedmail>.
- [7] Habeas, *Habeas Safe List*, <http://www.habeas.com/en-US/Senders/Safelist/>.
- [8] Habeas, *Habeas SenderIndex*, <http://www.habeas.com/en-US/Receivers/SenderIndex/>.
- [9] Brondmo, Hans Peter, Margaret Olson, Paul Boissonneault, *Project Lumos: A Solutions Blueprint for Solving the Spam Problem by Establishing Volume Email Sender Accountability*, 2003.
- [10] Jakobsson, Markus, Steven Myers, *Phishing and Countermeasures, Understanding the Increasing Problem of Electronic Identity Theft*, Wiley, 2006.
- [11] Microsoft Corporation, *Sender ID Framework – Executive Overview*, 2004.
- [12] Goodrich, Michael T., Roberto Tamassia, Danfeng Yao, "Accredited DomainKeys: A Service Architecture for Improved Email Validation," *Proceedings of the Second Conference on Email and Anti-Spam (CEAS)*, 2005.
- [13] Peterson, Patrick, "SIDF and DKIM overview Scorecard," *Authentication Summit II*, 2006, http://www.aotalliance.org/summit_archive/pdfs/2_Summit_Scorecard_final.pdf.
- [14] Proofpoint, *High-performance Email Reputation and Connection Management*, March, 2007, <http://www.proofpoint.com/products/dynamic-reputation.php>.
- [15] Price, W., *Inside PGP Key Reconstruction, A PGP Corporation White Paper*, 2003.
- [16] Ramachandran, Anirudh and Nick Feamster, "Understanding the Network-level Behavior of Spammers," *Proceedings of ACM SIGCOMM*, Pisa, Italy, 2006.
- [17] Realtime Blackhole List, Mail Abuse Prevention System LLC, California, 2002, <http://www.mail-abuse.org/rbl/>.
- [18] Sender Score Certified, *Return Path Management*, <http://www.senderscorecertified.com>.
- [19] Return Path, *Sender Score Email Reputation Management*, <http://www.returnpath.com/delivery/senderscore>.
- [20] Srivatsa, M., L. Xiong, L. Liu, "TrustGuard: Countering Vulnerabilities in Reputation Management for Decentralized Networks," *14th World Wide Web Conference (WWW 2005)*, Japan, 2005.
- [21] Jordan, Stephanie, Matt Blumberg, Des Cahill, Richard Gingras, "Accountable Email: Building on Authentication," *Authentication Summit II*, 2006, http://www.aotalliance.org/summit_archive/pdfs/7_building_on_authentication.pdf.
- [22] Wong, M. W., *Sender Authentication: What To Do*, Technical Document, 2004, <http://www.open-spf.org/whitepaper.pdf>.
- [23] Yahoo Inc., *DomainKeys: Proving and Protecting Email Sender Identity*, <http://antispam.yahoo.com/domainkeys>.
- [24] Yu, Bin and Munindar P. Singh, "An Evidential Model of Distributed Reputation Management," *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 2002.
- [25] Yu, Bin and Munindar P. Singh, "Detecting Deception in Reputation Management," *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, Melbourne, ACM Press, 2003.
- [26] Yu, H., M. Kaminsky, P. B. Gibbons, and A. D. Flaxman, "Defending Against Sybil Attacks via Social Networks," *Proceedings of ACM SIGCOMM Conference*, 2006.
- [27] Zimmermann, P., *The Official PGP User's Guide*, MIT Press, Cambridge, 1995.

Moobi: A Thin Server Management System Using BitTorrent

Chris McEniry – Sony Computer Entertainment America

ABSTRACT

We describe a tool that provides a method for running dataless caching clients – a hybrid combination of imaging systems with traditional diskless nodes. Unlike imaging systems, it is a single boot to get to a running system; unlike diskless systems, it is more robust and scalable as it does not continuously depend on central servers.

The tool, Moobi, uses the peer-to-peer protocol BitTorrent to provide efficient distribution of the image cache, and combines standard diskless tools to provide the basis for the running system. Moobi makes it possible to run large installations of “thin server” farms.

Introduction

Diskless clients are appealing for large clusters of similar systems performing similar functions since they limit the capability for local configurations and, more specifically, local misconfigurations on each system. This provides a more uniform environment and allows for the management of a large number of systems with fewer resources. The key for these systems is not so much that they are diskless in that they are dataless – there is nothing which is inherently tied to the nodes. In the desktop realm, these are typically referred to as thin clients. In the server realm, “throw away” and “field replaceable units (FRUs)” are some common terms.

The difference between diskless and dataless hinges upon drive cost, application robustness over drive failure rates, and the need for local working disk. The size of drives has increased significantly, especially when compared to the size of operating systems and applications bases. Pinheiro, et al. [1] demonstrated that annual hard drive failure rates were between 2% and 10% depending on age and usage; even with 2%, in a large population of machines, there is an inherent need for applications to account for downed nodes and thereby make drive failure moot. In addition, a local drive provides a convenient location for cached data or temporary work space, and so they are included in most clusters regardless of management methods. These reasons push commodity servers towards having local storage and treating it as a large cache or work disk, but not as a true data disk.

Since these systems have drives, they also have a unique local instance of the operating system. Whether this operating system is installed via an imaging system, or installed via an automated installation mechanism is largely irrelevant since the result is an individual system instance with a unique instance of every part of that system. This individual identity is necessary for certain items – hostname, ssh keys – but is undesirable for other items – /usr software. Any configuration management

system for these setups must account and check for every piece individually to validate the consistency of the environment.

Truly diskless clients do not have this problem. Nothing is locally maintained, and a single instance, typically of an NFS server, is directly used for the environment. This provides the consistency, but loses in performance: diskless clients are very hard to scale to 100s of nodes. This limitation on scaling is due to their reliance on the large file servers that support the diskless nodes. An installed environment suffers a similar scaling issue whenever massive deployments have to happen in short order – initial spin ups, disaster recovery, or massive updates. Diskless client environments have a continuous scaling issue, while the installed environments have “impulse” scaling issues.

Moobi combines the consistency of diskless systems with the efficiency of imaged or installed systems. It achieves this by dividing the operating system image into a small configurable portion – the root or /etc – and a large fixed image – /usr – and by caching and sharing the large image via BitTorrent. The fixed image is maintained as a whole instance, so only one item needs to be validated against the source. Every time a node boots, or as often as is necessary for auditing purposes, it can scan the cache and downloads and updates the cache with any new image.

BitTorrent is used for validating and updating the image cache. Due to its nature, these two operations are inherently connected. In order to download and update, it must validate. It sees updating as the only logical followup after validating. In addition, BitTorrent allows for an update mechanism which does not require large monolithic file servers; thus it alleviates the bulk of the scaling issues for large installations. In addition, BitTorrent allows for multiple systems to distribute the image without much additional logic. This makes it easier to make the system most robust overall.

The mechanisms used in Moobi provide additional availability for applications running on Moobi

nodes. Not only does Moobi reduce the MTTR, but it also provides several hooks for system hardening. Periodic image checking provides a level of confidence in the sanctity of the software being used. In addition, the static file system images, such as the software image `/usr`, can be mounted with read-only permissions. Application stacks could be wrapped up in a similar manner to provide extra confidence in the stack.

Going to a hybrid solution also acts as a stepping stone. Most organizations grow organically from individual system instances to dataless systems. Hard drives tend to remain an item in new systems, and are not removed from repurposed systems.

Related Work

Given Moobi's hybrid nature, similar work falls into either the imaging and installation category or into the diskless client category, yet neither category completely describes it. Unlike imaging systems, it is a single boot to get to a running system; unlike diskless systems, it is less fragile as it does not continuously depend on central servers.

Several commercial and noncommercial imaging systems exist to date. Norton Ghost [2] is the industry standard for commercial system imaging software. Several noncommercial systems provide similar features to Ghost, and have been able to address many of the distribution issues. Frisbee [3], in particular, has been shown to have very good image distribution performance [4]. Partition Image [5] provides a wide range of image support, but is not very efficient in its distribution.

Typical system imaging packages use one of two network mechanisms for the transporting of the golden image to the target machine: a unicast transfer, or a multicast or broadcast blast. Unicast transfers can be very expensive on the boot servers. Broadcast is very stressing on the network. Multicast requires advanced synchronization techniques and places additional constraints on how and when nodes can come online. In contrast, a BitTorrent swarm limits server load, limits the scope of network usage to the ports involved, and automatically handles synchronization – if a BitTorrent client is late to the swarm, its peers will catch it up.

All imaging systems require some mechanism to address host specific file overlays. Typically, this is done by just copying down the necessary files from a central system. Ghost in particular is very good at performing post imaging instance fix-ups to reduce unique identifier conflict without needing a central software repository. However, it does not address non-windows operating systems or other post installation fix-ups, and expects the environments to fix it either manually, or via a larger control system – Active Directory for instance.

By far, the prominent implementation of diskless linux is the Linux Terminal Server Project [6]. However, diskless nodes have a long history under other

operating systems, and also have a long history of performance issues with a central server [7]. Traugott and Huddleston [8] highlight Sun Autoclient's CacheFS features to reduce this. Moobi reduces these issues to occurring only at boot time, and allows for redundant boot servers with little or no synchronization between them.

BitTorrent is heavily in use in distributing large software images to large audiences. Many linux distributions, live CDs, large software applications, and VMWare appliance images [9] are distributed using BitTorrent. There are discussions of integrating BitTorrent with linux distributions' installation or update systems, but this discussion has produced little.

Recently, SystemImager [10] of the System Installation Suite has incorporated BitTorrent as a distribution method for its imaging system. Moobi and SystemImager run into many similar obstacles, especially when handling image versus host specific overlays, and when attempting to achieve high performance and scalability. However, SystemImager still views the many systems as single instances and can fall into divergent configurations. In addition, SystemImager has a two boot imaging operation – one boot for the imaging, and another boot to start the imaged system for normal operation.

Moobi Design

Moobi Basics

Several aspects are assumed for the Moobi image building and deployment discussion: how Moobi leverages the linux boot process, how Moobi distributes its own software, how additional configuration information is passed to identical nodes, and host specific overlays.

First, Moobi intercepts the normal linux boot process to prepare a specialized environment prior to handing off to the standard init process.

Almesberger [11] describes the linux boot process in great detail. The relevant portions for this discussion are summarized here. With loadable modules being as prominent as they are, almost all recent linux boots require the use of the `initrd`. Typically, the `initrd` is a pseudo-root file system which contains the kernel module utilities as well as any modules necessary for the system be able to mount the true root file system. If this is a local disk, disk drivers and file system drives are loaded. If this is a network file system, the NFS modules are loaded. Once the kernel is done with its primary initialization, it mounts the `initrd` as a ramdisk and executes the `linuxrc` script contained therein. Typically, the `linuxrc` script is simplified to the point of just loading the modules necessary and then hands off execution to `init`. Each `initrd` is built during any kernel installation or upgrade – the build process tailors it for the system at hand.

Moobi, like LTSP and many other alternative boot systems, uses the `initrd` and `linuxrc` as its springboard. An `initrd` for a Moobi booted system ends up being the full root file system. The `linuxrc` detects the

hardware coarsely, and initializes the network interfaces. It then connects to the boot server and retrieves the necessary torrent file for its image, starts up BitTorrent, and joins the swarm in progress. After the image is complete, it downloads the host specific overlay, then cleans up after itself and hands off normal boot execution to init.

For this to work, the kernel image and the initrd must be available to the booting node. The bios's PXEBOOT is used to retrieve pxelinux [12] which in turn retrieves the kernel image and initrd. In this case, pxelinux is the boot loader Almesberger references, and is responsible for placing the initrd into the ramdisk and then handing off to the kernel.

Moobi maintains a self contained instance of its software on each node during boot. This self contained instance is kept under /tmp/bootbin and /tmp/bootlib. By making it self contained, Moobi is removed from most dependencies on distribution resources. However, there is still a minimal expectation on the remainder of the root file system, such as standard shell and bin utils. Moobi cleans up this instance after the pre-init setup to avoid taking up precious resources during boot.

The ability to provide additional configuration or status information to the linuxrc during boot is very minimal. Basic identity – hostname and networking information – is provided by DHCP and DNS. Most other configuration aspects can be derived from those, or derived from hardware detection. However, some information must be passed in before those. For instance, unless they are autonegotiated, link speed and duplex must be known prior to network initialization; otherwise, none of it works.

Kernel variables are being used to pass in the additional configuration information needed for the bootstrapping step. MOOBI_NET is the parameter for the network link speed and duplex settings example above. MOOBI_NET can be set to AUTO, or 1000F, 100F, or other appropriate values. For the author, this was necessary due to some hardware and software compatibility issues related to network link autonegotiation. Additional parameters could be used if special parameters were needed to be passed for the storage subsystem startup, but a general hardware configuration mapping would be more useful. See Future Work for more discussion on this.

As is the case in most environments, the same image may be used in several slightly different contexts. For instance, the same software image may have different services enabled for it; or the same image may be used in multiple locations and needs to be customized for those locations – different network layer permissions such as firewall rules or tcpwrapper configs. While some of this can be handled via standard and optional extensions to DHCP, DHCP is not necessarily the best place to transfer file data. Almost every

imaging system uses host overlays, and Moobi is no different. For minor configuration differences between nodes using the same image, Moobi maintains a simplified file transfer over HTTP to retrieve the appropriate overlays. Other mechanisms could easily be swapped in for this.

Before a node is available for Moobi, it must be prepared with a disk partition layout which works for the images in use. The partitioning scheme only need be done once per node for any family of images that use that partitioning scheme. The only requirement for the partitioning scheme is that an image cache partition must exist. More advanced logic in the linuxrc could be used to overcome this limitation of the current implementation.

Moobi Image Build

A Moobi “image” is actually comprised of several file system images, file system skeletons, and the kernel file. One of the true file system images is the root file system. Typically, the other true file system image is /usr. Given most installations, this is the largest and single most monolithic image that does not change. A file system skeleton is a simple text file which summarizes file and directory x ownership and permission properties for a file system. This is useful for the variable file systems such as /var and /tmp. Since these will either be local disk or ramdisk, a pure imaging technique provides no advantage over a simplified technique which just ensures the existence of the structure.

Images can be transferred in one of three ways: TFTP – primarily just for the root file system image and the kernel file; HTTP – for the skeleton images; and, the preferred, BitTorrent – for the large file system images. The root partition is transferred via TFTP; since TFTP is the least efficient and robust, any transfers using it should be minimized, and therefore the root partition should remain as small and streamlined as possible. Luckily, linux kernels at this point recognize gzip compressed initrds, so the root partition can be compressed. The skeleton files are transferred via HTTP since it provides a reasonable ability to recover from transient system or network performance failures, and due to the fact that multifile torrent support is limited in some BitTorrent client implementations. Given the average size of a skeleton is on the order of 10K, a swarm is not necessary. BitTorrent is used for large file system images since it provides both a robust and efficient transfer mechanism and a native file hash checker.

The “imagebuilder” script has been developed which aids building images from scratch. Imagebuilder takes many of the standard build system inputs: a description of the file system containers or image files, pre and post installation scripts, a package list, and a miniroot of static file assignments. In addition, Imagebuilder takes a series of configuration options for the script run itself: source, destination and

working directories, the kernel package to be used, and the image name. The `imagebuilder.conf` uses the `.ini` configuration file format. A typical file system configuration looks like:

```
[partition:/]
name=root.img
size=128M
compress=1
skipmd5=1

[partition:/usr]
name=usr.img
size=1024M
compress=0
skipmd5=1

[partition:/var]
name=var.skel
skeleton=1
compress=0
skipmd5=1
```

Of note is the skeleton flag which signifies that the file system in question is a skeleton file system.

The imagebuilder process consists of:

1. Creating a temporary working directory to act as a installation root.
2. Running the pre-script to initialize the working directory.
3. Locating, via a search path, and installing all of the packages identified by the package list.
4. Copying over any additional stand alone files from a mini-root. The `linuxrc` is typically kept and installed from here.
5. Running the post-script on the working directory for any last fix-ups.
6. Building the image partition files, starting with the deepest path first and working up to the root.

For a true file system image, it creates the image file by:

- 6a. Generating an appropriately sized zeroed image file.
- 6b. Creating an ext2 or ext3 file system on the image file.
- 6c. Mounting the image file on a loopback device.
- 6d. Rsyncing the appropriate subdirectory onto the image file's mount point. Subdirectories for other images are explicitly excluded.
- 6e. Unmounting the image file and performing any additional operations on the image (generating a checksum hash, compressing it, etc.).

For skeleton images, it creates a text file by recursing through the appropriate subdirectory and collecting file name, ownership, and permission information.

This process does not need to be followed for a "valid" image to be built. Any generation system which creates valid image files containing valid file systems could be used. For instance, a linux live CD image could be used as a followup. However, for this

to work, the `linuxrc` must be able to retrieve, place, and open the image files.

Moobi Image Deployment

Once the image is built, it is moved to the boot server, broken up and located as appropriate for the image transfer mechanism. The kernel image and `initrd/root` file system are placed in the TFTP directory. The skeleton files are placed in the HTTP service's document directory. Large file system images are also located in the HTTP service's document directory, since a specific location is not necessary as the BitTorrent client will work anywhere. It also allows for singular fix-up transfers to happen over HTTP should the need arise.

The image shepherd, or `ishepherd`, process is responsible for maintaining all of the subsystems which are necessary for a complete Moobi boot: any network configurations, the `dhcp` daemon, the `tftp` daemon, the `http` daemon, and any BitTorrent seeders. The boot server is the primary for all of these, but additional boot servers could provide equivalent services. All function independent of each other, and typically, the first to respond would be the one to be used.

A typical Moobi node boot proceeds as shown in Figure 1.

The current boot server handles multiple VLANs, primarily to directly serve DHCP with network helpers. Since it already has an interface on each VLAN, it has a specific seed for that VLAN. This restricts network traffic two-fold. No BitTorrent traffic traverses routed interfaces, and this reduces the need for firewall holes. In most but not all cases, VLANs do represent separate network localities. Restricting network traffic to the VLAN should keep network stress localized. However, it would not require much additional configuration if an environment wants to allow routed interface traversing – does not want to provide a trunk to the boot server, does not have give VLAN distribution, etc. Each image distribution is given a specific port which is configurable and can be kept to a small range – so the firewall or network ACL changes could be kept minimal.

Experimental Data

Initial deployments have been able to boot over a hundred systems with 1.4 GB `/usr` images with an average deployment time of around 5-6 minutes on a 100 Mb/s ethernet network spanning multiple switches linked by multigig trunks. The boot failure rate with this high of a load has been unacceptably high at about 5-10% in the worst case which has been primarily in the initial tftp transfer.

A more rigorous set of tests were run for this paper. The environment consisted of 64 booting nodes, a DHCP/PXE boot server, and 1-4 BitTorrent seeds or one HTTP server. All systems were running on commodity hardware with a single 2.4 GHz P4, 4 GB of RAM, and a 1 Gb network connection. All devices

were evenly split across two Cisco 4948 switches with a 4 Gb trunk connecting them; logically, all devices lived on the same VLAN.

The runs were broken up into different load sizes (1, 4, 16, or 64 nodes), and into different load mechanisms or configurations. The configurations were labeled:

- SEED** BitTorrent distribution using a single seed, and the booting node continued sharing after it finished booting.
- NOSEED** BitTorrent distribution using a single seed, but the booting node stopped sharing after it finished booting.
- 4SEED** BitTorrent distribution using four seeds, and the booting node continued sharing after it finished booting.

- 4NOSEED** BitTorrent distribution using four seeds, but the booting node stopped sharing after it finished booting.
- HTTP** Download of the fixed image using wget over http.

Each series of runs – load size and configuration – was performed five times. For the single node boot, only three configurations were used: SEED, 4SEED, and HTTP. The NOSEED and 4NOSEED configurations are identical to the SEED and 4SEED configurations, respectively.

A run consisted of issuing a shutdown/restart to all of the nodes at the same time, and measuring all times relative to the shutdown issuance. Time measurements were taken from time of boot, time of the

Client Node	Server or Swarm
BIOS initialization	
DHCP request	
	DHCP response with network and followup up location for TFTP
TFTP request	
	TFTP response of pxelinux
TFTP request for kernel image and initrd	
	TFTP response of kernel and initrd
initrd loaded into ramdisk and kernel loaded	
kernel initialization	
kernel execs linuxrc	
linuxrc hardware detection	
network reinitialization DHCP request	
	DHCP response with network information
self identification DNS reverse lookup	
	DNS response
Mount cache file system	
HTTP request for image torrent	
	HTTP response of image torrent
BitTorrent Client startup image file checksum join swarm	
	Swarm sending and receiving image data
	Image file finish
Mount image file as appropriate	
HTTP request	
	HTTP response of skeleton files
buildSkeleton for each skeleton file	
Multiple HTTP requests	
	HTTP responses of overlay host files
Shutdown network and cleanup	
Hand off to init for normal boot	

Figure 1: Boot Times

fixed image transfer start, time of the fixed image transfer end, and time of the end of boot(last command in rc.local). From these the time for PXE transfer and fixed image transfer were calculated and reported; see Table 1.

Node Cnt.	Method	Total	PXE	D/L	Times(s)	Failures
1	SEED/NOSEED	397.8	27.4	166.4	0.0	
1	4SEED/NOSEED	390.8	27.2	163.0	0.0	
1	HTTP	245.0	27.2	19.0	0.0	
4	SEED	452.8	27.0	226.8	0.2	
4	NOSEED	450.8	27.4	221.0	0.0	
4	4SEED	418.6	30.0	188.8	0.0	
4	4NOSEED	419.8	29.4	186.6	0.2	
4	HTTP	277.0	28.7	66.3	0.0	
16	SEED	542.8	36.0	315.0	0.0	
16	NOSEED	490.2	36.2	258.8	0.0	
16	4SEED	443.4	35.4	204.0	0.2	
16	4NOSEED	456.8	35.6	225.6	0.2	
16	HTTP	449.5	36.2	233.0	2.2	
64	SEED	602.2	127.6	280.2	0.0	
64	NOSEED	623.8	117.4	313.0	2.6	
64	4SEED	569.0	122.6	247.6	0.2	
64	4NOSEED	626.2	109.8	320.6	1.4	
64	HTTP	1208.0	135.3	892.0	0.0	

Table 1: Execution times.

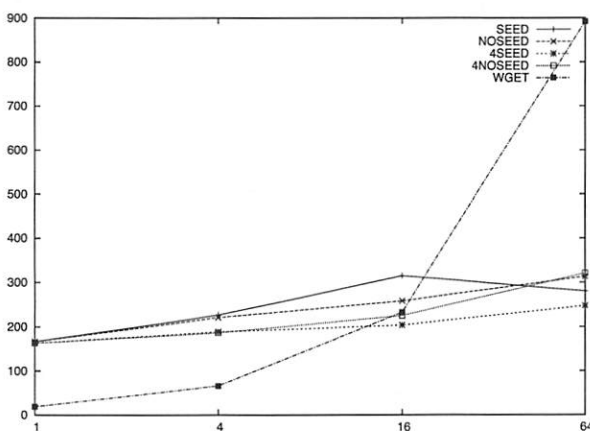


Figure 2: Boot times.

CPU and network usage for the serving hosts were watched during the course of these runs. Of note, only two resources appeared to be stressed:

1. the HTTP server's network interface during the fixed image transfer – expected since it is the only source for upwards of 128GB of data to transfer.
2. the DHCP/PXE boot server's load during tftp transfers – it spawns separate processes for each tftp connection.

Beyond those, CPU utilization never passed 35% and network utilization never passed 10%.

A kickstart was also performed as an additional comparison. The approximate time for the kickstart

through to an available system was on the order of 1200 seconds. Given the distribution method for the kickstart(single source HTTP), the expected time would increase as the HTTP methods from above.

Observation 1: As expected, a peer-to-peer distribution system works exceedingly well in scaling to many nodes.

Observation 2: The transfer time for the PXE portion grows as a single source. This leads to boot failures which require manual intervention.

Observation 3: There is only a minor improvement when providing additional seed nodes.

Observation 4: The time for a seed image distribution compared to an installed mechanism was approximately 1:4.

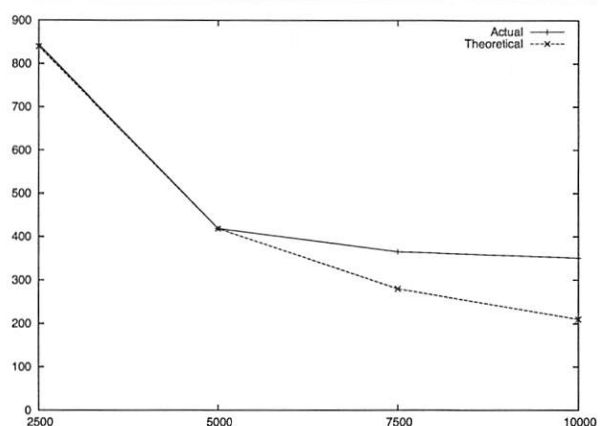


Figure 3: Boot Times with a Rate Limited BitTorrent.

Observation 5: The largest influencer of distribution time is the initial seed's transfer rate. Even for a large environment – 64 nodes – the download time was very close to the case where one node would download at the transfer rate.

Transfer Rate Limit	Actual Time			Theoretical Time
	Total	PXE	D/L	
2500	1165.6	122.8	844.2	839.7
5000	740.7	122.3	419.0	419.0
7500	682.4	123.4	365.4	279.9
10000	665.5	120.5	350.8	209.9

Table 2: Boot Times with a Rate Limited BitTorrent.

Summary

In general, Moobi performs very well for large installations. It appears to have a break point on the order of 10s of nodes at which it is equivalent to installed or imaged systems from a single node. In addition, it does not show significant performance degradation as the number of clients increases. Given its efficient scaling, it allow for large clusters of systems which are image updated on a very regular basis, and this allows for a shift in the way those systems are managed.

Critique

Moobi is reliant upon the node's bios's pxe implementation, either system or NIC, for proper behavior. Several pxe implementations which the author has worked with are not very robust. They either are unable to recover from an overloaded boot server to hand out a response in a given unconfigurable timeframe, or they are unable to recover from a slow tftp transfer – once the tftp has slowed down, the client will not receive data any faster than the lowest previous data send even if the server is able to recover and begin sending more. Additional boot servers, a more efficient tftp server, or better bios support could overcome this.

Several pieces of the current Moobi implementation are specific to RedHat-like distributions. This is just a limitation of this implementation and not a limitation of the technique. RPMs could be replaced with DEBs. The python BitTorrent client and related host-files tools could be replaced with compiled or other scripting languages better suited for the desired distribution.

Currently, the linuxrc used within Moobi is very specific to the structure of the image in use. This is a feature and failing of Moobi since it requires in depth knowledge of the process to be able to boot systems. This could be made to be more robust and accept different image structures as an output of the image building process.

Future Work

Advanced hardware detection. Currently, this is only variable to three possible hardware configurations and so the detection mechanism is very minimal – mainly observing the hard drive (IDE versus SCSI) and the ethernet interfaces. This fails to scale as is, but two approaches can be attempted for this. The first is a relatively small hardware lookup table. Since many large organizations use a limited set of vendors and system configurations, a small table will most likely be sufficient. This could be passed in as a kernel command line parameter (MOOBI_HARDWARE_ID or so), or a similar detection of base hardware assets.

Image overlays. The image overlay system is very immature. It currently just lists files to be transferred with the correct properties. This would be an ideal place to drop in high level configuration management systems, such as Cfengine or Puppet, for a more manageable approach.

Partial image updates. The current usage of Moobi treats each new image as a completely different item. This means that all updates involve transferring the entire new image, and Moobi has certainly been optimized for this. However, an additional incremental improvement can be achieved when acknowledging that most of the time the primary image is not very different. Typically, only a small fraction of the image description changes – a new software package or such

– and that corresponds to only a small fraction of the fixed image file changing. Given BitTorrent's blocking scheme, it would pick up that only the blocks with changes need to be sent, and could easily reduce the image transfer.

BitTorrent locality information. BitTorrent is very efficient, yet it does not know anything about its local world. All clients are equivalent to it. Since many large cluster nodes are grouped by switch, it would be convenient to leverage this so as to not saturate any switch to switch links.

Author Biography

Chris McEniry left the Massachusetts Institute of Technology with a BS-EECS in 2000, and has spent over ten years as an official and unofficial systems administrator. He can most recently be found in Southern California, helping to keep the lights on with Sony Computer Entertainment America (aka Playstation). Reach him electronically at cmceniry@mit.edu.

Bibliography

- [1] Pinheiro, Eduardo, Wolf-Dietrich Weber, and Luiz Andre Barroso, "Failure Trends in a Large Disk Drive Population," *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pp. 17-28, 2007.
- [2] Norton Ghost, http://www.symantec.com/home_homeoffice/products/overview.jsp?pcid=br&pvid=ghost12.
- [3] Hibler, Mark, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb, "Fast, Scalable Disk Imaging with Frisbee," *USENIX Annual Technical Conference Proceedings*, pp. 283-296, 2003.
- [4] White, Brian, et al., "An Integrated Experimental Environment for Distributed Systems and Networks," *Proceedings of the 5th Symposium on Operating Systems Design & Implementation*, pp. 255-270, 2000.
- [5] Partimage, <http://www.partimage.org>.
- [6] McQuillan, J., "The Linux Terminal Server Project: Thin Clients and Linux," *Proceedings of the 4th Annual Linux Showcase and Conference (ALS2000)*, Usenix Association, 2000.
- [7] Gusella, R., "A Measurement Study of Diskless Workstation Traffic on an ethernet," *IEEE Transactions on Communications*, Vol. 39, Num. 9, pp. 1557-1568, 1990.
- [8] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the 12th Systems Administration Conference (LISA '98)*, pp. 118-196, 1998.
- [9] BitTorrent download info, <http://torrent.vmware.com:6969/>.
- [10] Finley, Brian, "VA SystemImager," *USENIX Annual Linux Showcase and Conference Proceedings*, pp. 181-186, 2000.

- [11] Almesberger, Wener, "Booting Linux: The History and the Future," *Proceedings of Ottawa Linux Symposium*, 2000.
- [12] *PXELINUX – SYSLINUX for network boot*, <http://syslinux.zytor.com/pxe.php>.

PoDIM: A Language for High-Level Configuration Management

Thomas Delaet and Wouter Joosen – Katholieke Universiteit Leuven, Belgium

ABSTRACT

The high rate of requirement changes make system administration a complex task. This complexity is further influenced by the increasing scale, unpredictable behaviour of software and diversity in terms of hardware and software. In order to deal with this complexity, configuration management solutions have been proposed. The processes that many configuration management solutions advocate are kept close to manual system administration. This approach has failed to address the complexity of system administration in the real world. In this paper, we propose PoDIM: a high-level language for configuration management. In contrast to many existing configuration management solutions, PoDIM allows modeling of cross machine constraints. We provide an overview of the PoDIM notation, describe a case study and present a prototype. We believe that high-level languages are needed to reduce system administration complexity. PoDIM is one step in that direction.

Introduction

The fact is that configuration errors are the biggest contributors to service failures (between 40% and 51%). Configuration errors also take the longest time to repair [37, 36, 34]. As the complexity of computer infrastructures increases, the risk of configuration errors increases likewise and introduces even higher change costs. Changes to a configuration can be technically – such as software upgrades – or business oriented. A difficulty with configuration changes is the high number of dependencies between systems. Systems do not operate in isolation, but in a network. A change in the configuration of one networked service may cause a complex chain of changes in dependent services. Furthermore, infrastructural complexity is influenced by increasing scale, unpredictability in software behaviour and systems variety [21, 39, 4].

1. **scale:** The number of network devices, servers, desktops and laptops in a typical infrastructure is increasing significantly. New kinds of devices such as PDA's, mobile phones and sensor nodes are extending the scope of an organizational computer infrastructure.
2. **unpredictability in software behaviour:** Increasingly complex software systems tend to have more bugs, viruses and vulnerabilities. Bugs in software, viruses and vulnerabilities make full control over the system's behaviour an illusion [13].
3. **systems variety:** Computer infrastructures have a large variety in terms of hardware platforms, operating systems and application software. Our definition of infrastructures includes not only desktops, servers and laptops, but also embedded devices such as palmtops, mobile phones and network devices such as routers and switches. All of these devices run on a variety of operating systems and accompanying application software.

Using a network shell or a configuration management language whose process is close to manual system administration simply does not work in large and varied computer infrastructures with complex software systems. Indeed, the subtle interactions between (different versions of) software packages can make systems with the same operating system and hardware platform unique. According to [5], the cost per unit becomes excessively large when using manual management processes. More loose, higher-level, processes are necessary.

PoDIM abstracts from systems variety and allows, more than existing configuration management languages, expressing an administrator's intentions. Expressing intentions is clearly of a higher-level nature than expressing, for example, what lines in the `sendmail.cf` file need to be modified on a mail relay. The key concept of PoDIM's high-level language is that it allows modeling of cross machine constraints.

The remainder of this article is structured as follows. First, we introduce PoDIM as a high-level language for configuration management. Next, We elaborate on PoDIM in the sections on Configuration Descriptions and Rules. The run-time semantics of PoDIM are introduced in the Prototype Section. We also present a case study. We end with sections on related and future work.

Language Overview

The state of the art in configuration management allows assigning roles to machines and setting high-level parameters for those roles. "Configure machine X to be a web server" and "configure machine Y to be DHCP server" are examples of role assignments. "The web server must run on port 80" is an example of a parameter assignment. The PoDIM language aims for a higher level of abstraction. Instead of role assignments, we want to express things such as: "One of my

servers must be configured as a web server” and “On every subnet, there must be two DHCP servers.” Instead of parameter assignments, we want to express things such as: “A web server must use a port number higher than 1024.”

PoDIM’s language consists of a rule language and a domain model. The distinction between domain model and rule language is a recurring theme in policy languages. A domain model provides a description of the domain in which a rule language solves problems. Since we are dealing with the domain of configuration management, the domain model contains descriptions for things such as DHCP servers and web servers. The rule language defines types of rules and how they interact with the domain model. A system administrator writes rules in PoDIM’s rule language. These rules interact with the domain model and output a configuration for each managed device. The next section elaborates on the domain model. Subsequently, we describe PoDIM’s rule language.

The basic principle of PoDIM’s runtime is that each real world object is simulated in the system. The different classes of objects are defined in the domain model. Examples of object classes are devices, network interfaces and services such as DHCP servers and web servers. PoDIM’s runtime takes a set of policy rules as input and tries to satisfy these rules by creating objects and setting parameters of objects. In doing so, it generates a configuration for each managed device. Existing tools, such as Cfengine [12, 10, 11, 14], can then be used to deploy the generated configuration on each real world device.

Configuration Descriptions

PoDIM’s domain model is object-oriented. This means that the “things” in the domain model are coded as classes. Examples of classes are DHCP server and web server. We use an existing object-oriented programming language for coding classes, named Eiffel [30, 31].

Figure 1 shows a simplified graphical representation of the domain model in the BON notation [40]. All classes, such as DHCP server and web server, have one common ancestor: ENTITY. The ENTITY class is used for modeling common functionalities. The arrows denote inheritance relationships. The simplified example presented in Figure 1 uses only single inheritance relationships. In real life examples, multiple inheritance is often necessary. Eiffel supports this.

Classes define the interface of a software subsystem in PoDIM. A class has attributes that can be set, queries that can be executed and commands that can be executed. For example, the WEB_SERVER class has an attribute for setting its port, a query to find out the administrative mail address and a command to enable php support on the server. Attributes are set by the system administrator when writing rules. Queries are used by the system administrator and other objects to gather information about the runtime system. Commands are used as an inter-object communication mechanism. An example of the latter occurs when a webmail object commands a web server to enable php support.

In the rest of this section, we elaborate on the definition of classes. We start with attributes and queries. Attributes are an object’s data structures. Queries define the questions one can ask an object. Next, we discuss commands. Commands define how objects can change each other’s state. We end this section with a description on how dependencies are modeled between classes.

Attributes and Queries

Attributes define the data structures for objects of a class. Queries are methods which return a result. In Eiffel, all attributes are also queries by definition, i.e., objects can query each other’s attributes. An object can only modify another object’s attributes by using commands. The result of a query is computed based on the results of other queries or the values of attributes. The example in Listing 1 shows a partial web server class. It defines two attributes: “php_supported” and “domain”

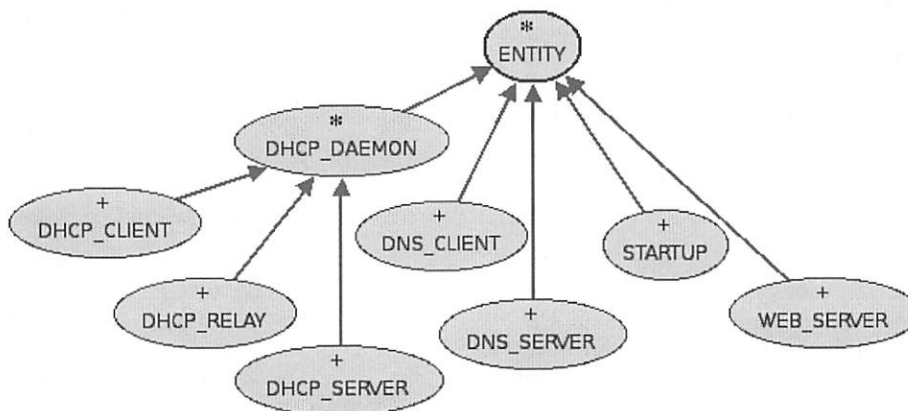


Figure 1: Domain model in BON [40] notation. All units of functionality such as mail servers and DNS clients are modeled as classes. All classes have one common ancestor: ENTITY. The arrows denote inheritance relationships.

and one query: “administrator_email”. In the example, the “administrator_email” query is based on the attribute “domain”. All attributes and queries have a type. For example, the “php_supported” attribute has type BOOLEAN. Note that the definition of WEB_SERVER is used as an illustration, not as an introduction to a real world WEB_SERVER class.

Commands

Commands are methods that change the state of an object (i.e., modify its attributes), but do not return a result. The example in Listing 1 contains a partial web server class with one command: “enable_php”. The “enable_php” command changes the value of the “php_supported” attribute. Since a command can contain arbitrary code, its behaviour should be clearly documented. For this documentation, we use another feature of Eiffel: preconditions and postconditions. Preconditions express conditions that need to be true before the command is executed while postconditions express the effects of the command’s execution. In our web server example, the precondition of the “enable_php” command is that php support is not yet enabled. Its postcondition expresses that php support will be enabled when the command is executed.

It is also possible to control access to commands, i.e., prohibit objects to execute commands on other objects. In the web server example, we could only allow objects that run on the same device to enable php support. In the Section on authorizing commands, we elaborate on how to specify access controls for objects.

Modeling Dependencies

A lot of dependencies exist between classes (and their real world software configurations). Imagine a startup class that is responsible for generating the

/etc/init.d directory on Linux systems.¹ Both the web server and DHCP server classes depend on the startup system. Indeed, if these two network services have no hook into the startup system, they are not activated when we reboot a machine. Another example of a dependency is the relationship between the implementation of a service and its attributes. Imagine a web server class that supports two web server implementations: apache and publicfile. Apache supports php, publicfile does not. In this case, php support can never be enabled on a web server object if it uses publicfile as its implementation.

To make these kinds of dependencies explicit in our domain model, we use two language constructs of Eiffel: references and invariants. When declaring an attribute in a class, it will contain a reference to another object and not to the contents of the actual object. For example, the dependency of the web server on the startup system is modeled as a reference in Listing 2 on line 9. Invariants are arbitrary boolean expressions that are required to be true at all times during an object’s lifetime. They provide a built in mechanism for modeling fine grained dependencies (and other restrictions on an object’s attributes state). For example, the relationship between php support and the chosen implementation is modeled in Listing 2 on line 12.

Making dependencies that exist in an IT infrastructure explicit in the domain model has two advantages. First, they can be used as a documentation aid. Second, a dependency violation can be detected by the PoDIM runtime. For example, when one rule states that the attribute “implementation” of a web server class must be set to “publicfile” and another rule

¹Classes can be used to abstract away from details like the operating system used and different software versions. For example, the same startup object results in other files being generated on Linux and BSD systems.

```

01 class WEB_SERVER
03 feature -- Attributes
05     php_supported: BOOLEAN
07     domain: STRING
09 feature -- Queries
11     administrator_email: STRING is
12     do
13         Result := "webmaster@" + domain
14     end
16 feature -- Commands
18     enable_php is
19     require
20         php_supported = False
21     do
22         php_supported := True
23     ensure
24         php_supported = True
25     end
27 end

```

Listing 1: This partial WEB_SERVER class defines two attributes: “php_supported” and “domain”, one query: “administrator_email” and one command: “enable_php”.

states that the attribute “php_supported” must be set to true, a dependency violation is detected. The default behaviour is to signal an error and abort.

Rules

The rule language is the user interface for the system administrator. It defines rules for expressing how the configuration of an infrastructure must look like. Remember that each real world object is simulated with PoDIM. For example, there exists an object for each device in your system, each network interface and every service that needs to be configured. The domain model presented in the previous section is a static description of the possible classes that can exist in the system. The rule language is used to create and manipulate objects.

A distinguishing feature of our rule language is that it allows the specification of constraints. We demonstrate the need for constraints with two examples.²

- When configuring a web server, the port is one of the attributes that can be set. A constraint allows expressing things such as “the port should be set to 80 or a value higher than 1024.” In contrast, a regular assignment only allows expressing things such as “the port should be set to the value 80.”
- Servers typically have roles assigned which determine the services they must offer. For example, one can state that system X is going to be a web and mail server. By using constraints we can express things such as: “A device should not provide more than four network services,” “I want two DHCP servers on each subnet,” or “One of my servers should configure itself as a web server.”

Since the domain model is object-oriented, the rule language contains rules to create objects and modify the attributes of objects. Remember that a class also defines commands for its objects. Commands allow objects to change each other's behaviour. The rule language also allows access controls between objects to be defined. In the rest of this section, we

²Since this work is about configuration management, we use examples from this domain. Nevertheless, the rule language is generic enough to apply to other domains.

```
01 class WEB_SERVER
02   feature -- Attributes
03     implementation: STRING
04     php_supported: BOOLEAN
05     startup: STARTUP
06   invariant
07     implementation.is_equal("publicfile") implies not php_supported
08   end
```

Listing 2: This partial web server class illustrates the invariant mechanism to model the dependency between the implementation and php support and the reference mechanism to model the dependency between the web server and startup system.

elaborate on the three types of rules: creating objects, modifying objects and authorizing commands.

Creating Objects

As a system administrator, you configure the network to offer services. This results in assigning a set of roles to each device in the network. For example, machine A acts as a web server and DHCP server. Machine B acts as a DNS server. All machines act as IPv4 nodes or routers. PoDIM's creation rules express role assignments precisely. Since every real world object is simulated in the PoDIM runtime, rules must exist for all real world objects to be created. In general, a creation rule instructs a set of objects to create other objects. For example, we instruct machine A to create a web server and a DHCP server. How do we know that a simulated object of machine A exists in the system? We can not assume this, so we have to create it with a creation rule. But then again, which object needs to create machine A? To get out this bootstrapping problem, we assume the presence of one object of a predefined class called SYSTEM_ENTITY.

Listing 3 shows a creation rule to create machine A. The rule contains three parts: The first part on line 1 specifies the rule type. In this case, we want to write a creation rule. The rule type can be extended with an optional rule identifier, in this case “machine_A”. The second part states which object needs to be created. In this case, we want to create a DEVICE object (line 2). We also specify one initial attribute for the device object that is going to be created on line 3. The attribute “name” will be set to “machine_A”. The third and last part on line 4, after the “select” keyword, specifies which object(s) need to execute this rule. In this case, we want all objects of class SYSTEM_ENTITY to execute this rule. By definition, only one SYSTEM_ENTITY object exists, so this rule will only be executed by one object. In plain English, the rule in Listing 3 reads as “A device object with name machine_A must be created.”

```
01 creation machine_A
02   DEVICE
03     name "machine_A"
04   select SYSTEM_ENTITY
```

Listing 3: This creation rule reads as “A device with name machine_A must be created.”

Now that we can write rules to create objects for all managed devices, it is time to enable some functionalities on those devices. For example, all devices need to resolve host names to addresses. Consequently, we need to enable each machine's DNS configuration. To enable this, we assume the presence of a `DNS_CLIENT` class in the domain model. Listing 4 shows how to express that every machine should configure itself as a DNS client. The rule has `"dns_clients"` as its identifier (line 1). In this case, the object that needs to be created is of class `DNS_CLIENT` (line 2). The objects that need to create a `DNS_CLIENT` objects are all objects of class `DEVICE` (line 3). Enabling functionality on a device is thus equivalent to instructing a device to create an object that represents that functionality (in this case, a DNS client). In plain English, the rule in Listing 4 reads as "All machines must act as a DNS client."

```
01 creation dns_clients
02     DNS_CLIENT
03     select DEVICE
```

Listing 4: This creation rule reads as "All machines must act as a DNS client."

In many cases, a more fine grained mechanism is needed to describe which objects need to execute a rule. For example, how would you say that all machines you use as a server need to configure themselves as a web server? To enable this, the part of a rule that selects objects on which to apply the rule can be further refined with a boolean expression. This boolean expression filters the objects that apply the rule. For example, in Listing 5 the selection clause on lines 3-4 includes all `DEVICE` objects, except for those where the boolean expression on line 4 evaluates to false. In plain English, this rule reads as "Machines with label 'server' act as `WEB_SERVER`." Note that we assume the presence of a `"labels"` attribute in the class `DEVICE`. The contents of this attribute can be easily set when writing rules that create devices. This is done in the same way as we assigned the value `"machine_A"` to the `"name"` attribute in Listing 3.

```
01 creation mail_servers
02     WEB_SERVER
03     select DEVICE
04     where DEVICE.labels.has("server")
```

Listing 5: This creation rule reads as "Machines with label 'server' act as `WEB_SERVER`."

```
01 creation mail_service
02     WEB_SERVER
03     select DEVICE
04     where DEVICE.labels.has("server")
05     group by DEVICE.labels.has("server")
```

Listing 7: This creation rule reads as "One device with label 'server' must act as a web server."

```
01 creation constraint dhcp_servers
02     [ 2 : 2 ] DHCP_SERVER
03     select NETWORK_INTERFACE
04     group by NETWORK_INTERFACE.subnet_interfaces
```

Listing 8: This creation constraint rule reads as "Each subnet must have two DHCP servers."

The syntax of the select-clause – lines 3 and 4 of Listing 5 – is modeled after SQL `SELECT` statements [2]. The name of a table – class name in our case – follows the `"select"` keyword. The optional `"where"` clause excludes rows – objects conforming to the class name – where the boolean expression evaluates to false. All queries and attributes of a class can be used in a boolean expression. Operators are used to compose composite expressions. Listing 5 uses the feature call operator. Other examples of operators are: comparison operators, boolean operators and arithmetic operators.

In many cases, you want to express not only *what* objects need to be created on a `DEVICE` – or another object – but also *how many* need to be created. This is where creation constraint rules come into the picture. Listing 6 expresses the previously mentioned example that "a device should not provide more than four network services". Creation constraint rules have an extra keyword: `"constraint"`. The name of the class to be created is also prefixed with an interval. In this case the interval expresses that a maximum of four server objects can be created. Note that we are using the inheritance features from the domain model in this example. We assume that all types of network services such as DHCP servers and web servers inherit from the `SERVER` class.

```
01 creation constraint server_objects
02     [ 0 : 4 ] SERVER
03     select DEVICE
```

Listing 6: This creation constraint rule reads as "A device should not provide more than 4 network services."

Often, you don't care which `DEVICE` will be your web server, as long as one – or more – devices are configured as web server. This can be expressed with the `"group by"` clause of the SQL `SELECT` syntax. The group by clause applies a rule to a group of objects rather than to of single objects. Listing 7 then reads as "One device with label 'server' must act as a web server."

We end with an often cited example in the context of configuration management: "I want two DHCP servers on each subnet." The rule for this example is shown in Listing 8. This example combines constraint rules and rules with `"group by"` clauses.

Before we explain the rule itself, we introduce the `NETWORK_INTERFACE` class. In the same way as we can create devices, DNS clients and web servers objects, we can create objects representing network interfaces. It does not matter if an object represents hardware (such as device and network interface) functionality or software functionality (such as DNS client and web server). The basic concept is that the `SYSTEM_ENTITY` object creates `DEVICE` objects. `DEVICE` objects can be instructed to create other objects such as `DNS_CLIENT` or `NETWORK_INTERFACE` objects. In the same way, `NETWORK_INTERFACE` objects can be instructed to create `DHCP_SERVER` objects, which is the functionality demonstrated in Listing 8.

The interval on line 2 limits the number of `DHCP_SERVER` objects to two. The “subnet_interfaces” query of the `NETWORK_INTERFACE` object returns a set of all subnet interfaces in the same subnet as the object on which the query is executed. The result of the “select” clause on lines 3-4 will be a set of network interface sets. Each inner set represents one subnet. On each of those inner sets, the rule to create two `DHCP` servers is executed, which results in two `DHCP` servers on each subnet.

Modifying Attributes

Once roles are assigned to devices, you want to tune the behaviour of those roles. Your web server needs a port to run on, your `DHCP` server needs to know whether it should serve fixed addresses, your `DNS` client needs to know what its domain is, and so forth. These examples can be expressed with PoDIM’s attribute assignment rules. They change the value of an object’s attributes.

Let’s start with the simple case: how do we specify the search domain for our `DNS` clients? The rule that realizes this is shown in Listing 9. Rules dealing with attribute assignments are called assignment rules (hence the keyword “assignment” on line 1 of Listing 9). In general, an assignment rule consists of a series of attribute-value assignments that are applied to the objects in the select-clause. In our example, we show one attribute-value assignment, where the attribute is “search_domain” and the value is “mydomain.com”. The objects on which this assignment is applied are, in this case, all `DNS` clients.

```
01 assignment dns_search_domain
02     search_domain    "mydomain.com"
03     select DNS_CLIENT
```

Listing 9: This assignment rule reads as “All `DNS` clients have mydomain.com as their search domain.”

```
01 filter php_enabling
02     enable_php block
03     select ENTITY, WEB_SERVER
04     where not ENTITY.device.is_equal(WEB_SERVER.device)
```

Listing 11: This filter rule reads as “PHP support on web server can only be enabled by objects on the same device.”

In some cases, you don’t care what value an object’s attribute has, as long as it’s within a predefined range. For example, you might want to express that “the port of all my web servers should be set to 80 or a value higher than 1024.” This is where assignment constraint rules come into the picture. Listing 10 shows the assignment constraint rule for our example. The attribute to be set is called “port”. The valid values for this attribute are the union of the singleton 80 and all values greater than 1024.

```
01 assignment constraint webserver_ports
02     port      [ 80 ] + [ 1024 : ]
03     select WEB_SERVER
```

Listing 10: This assignment constraint rule reads as “A web server’s port must be within the range 80 or a value greater than 1024.”

Authorizing Commands

Many system administrators work in a team. In most teams, people have roles: Jack is our `Linux` server specialist, Greg is our networking guy and Bill is our desktop guy. In small teams, communication is easy – Jack, Greg and Bill are located in the same office. In larger teams, however, there is a need to specify roles more precisely and enforce those automatically.

Since Bill is our desktop guy, we do not want him to configure network services of any kind. How do we express this? Consider the `SERVER` class. All network services like `DHCP` servers and web servers inherit from this class. The `SERVER` class thus represents common functionality for network services. We want to express that Bill cannot modify the attributes of an object if it inherits from `SERVER`. To realize this, we first introduce two extra PoDIM features: a rule type to express access controls and support for writing rules about other rules.

Recall from the discussion of commands that we wanted to limit access to the “enable_php” command on a web server to objects that run on the same device as the web server. To allow this, we introduce a third type of rule: filter rules. Remember that we already introduced creation and assignment rules. Listing 11 shows a filter rule for the case where we want to limit access to the “enable_php” command of web servers. The filter rule blocks the execution of the “enable_php” command on a `WEB_SERVER` for every `ENTITY` that is not created by the same device as the `WEB_SERVER`.

Filter rules allow to block the execution of a command based on the caller object and callee. A filter rule starts with the “filter” keyword and has an

optional identifier. It contains one or more commands (with optional arguments) that need to be blocked. The selection part on lines 3-4 is a bit different than that of creation and assignment rules. A filter rule always selects pairs of objects to identify a caller/callee pair. In SQL terminology: the SELECT clause contains a join of tables named ENTITY and WEB_SERVER. The resulting tuple-set is then filtered with the “where” expression on line 4. In this case, we express that we want to block communications when the caller is any entity and the callee a WEB_SERVER (line 3). If the caller executes the “enable_php” command, it is blocked when the caller is not created by the same device as the web server.

The other feature we need are rules about other rules. When we want to express that Bill cannot configure network services of any kind, we need a filter rule that prohibits the modification of objects representing network services. The only way Bill can modify objects is to write assignment rules. So, we want to write a filter rule that blocks assignment rules written by Bill from being applied on network services. Remember that we defined a common class for network services in this example, called SERVER. Since a filter rule specifies a policy for the interaction between two objects and assignment rules are in this case part of the interaction, assignment rules themselves must be objects.

Let’s go into more detail on how rules can be objects themselves. Take for example the assignment rule from Listing 9. The assignment rule contains a rule identifier, “dns_search_domain”, an attribute that needs to be modified, “search_domain”, the value for that attribute, “mydomain.com”, and the set of target objects, all DNS_CLIENT objects. Looking at this rule as an object, we have an object with attributes “identifier”, “attribute”, “value”, and “targets”. In this example, the value of “identifier” is “dns_search_domain”; “attribute” is set to “search_domain” and so on.

Since rules exist as objects in our system, they must have a static definition (class) in the domain model. An updated graphical representation of our domain model is shown in Figure 2. PoDIM’s three type of rules – creation, assignment and filter – are shown as classes in the domain model.

We have now introduced all features needed for expressing that Bill cannot configure network services. This policy is represented with a filter rule shown in Listing 12. The filter rule deals with the interaction between ASSIGNMENT_RULE objects and SERVER objects. By definition, the “execute_assignment_rule” command is used to execute an assignment rule on an object. Since we want to forbid Bill to

configure network services, we must block the execution of this command for rules created by Bill on SERVER objects.

Note that we depend on the presence of the “creator” attribute of an ASSIGNMENT_RULE. This attribute can be set with another assignment rule. We will not delve into how we can be sure that identities can not be spoofed. For now, it suffices that this can be achieved with public key cryptography and rule signatures.

Notice from Figure 2 that rule classes have a common parent: RULE. RULE itself is a child of the common class MANAGED_OBJECT, as is the ENTITY class that was discussed previously. In the same way that you can not compare a DNS client with a web server, it is useless to compare rules with entities. They both have the same structure: they contain attributes, queries and commands, but they represent very different things: rules represent intentions on the part of the system administrators while entities represent real world objects such as devices, network interfaces and network services.

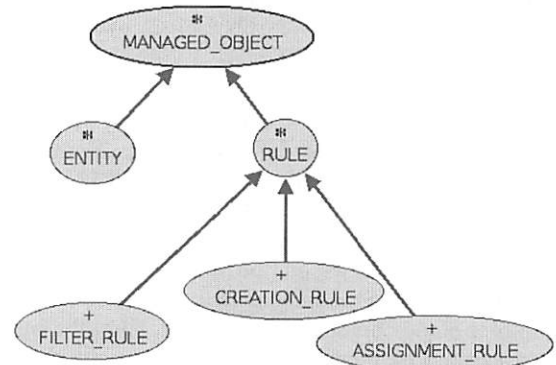


Figure 2: Domain model in BON [40] notation. This model includes RULE classes. RULE and ENTITY have a common parent: MANAGED_OBJECT. The arrows denote inheritance relationships.

Prototype

The prototype described below is available for testing from <http://purl.org/podim/devel>. First we describe the rule resolution process. Next, we describe how a configuration is deployed on a set of machines.

Rule Resolution

We have seen that the basic principle of PoDIM’s runtime is that objects are created for each real world “thing”. We have also discussed how a system administrator uses creation rules to specify which objects

```

01 filter bill_cannot_configure_services
02     execute_assignment_rule(rule) block
03     select ASSIGNMENT_RULE, SERVER
04     where ASSIGNMENT_RULE.creator = "BillsPublicKey"
  
```

Listing 12: This filter rule reads as “Bill cannot configure network services.”

need to be created. The basic form of a creation rule is that it states that an object or objects of a particular class must be created by other objects. For example, we can assert that all devices must create a DNS client object. Remember that there is a bootstrapping problem with this approach. To solve this, we assumed the presence of one object of a predefined class, `SYSTEM_ENTITY`.

The component responsible for creating a `SYSTEM_ENTITY` object is the translation controller. The translation controller contains compiled versions of all domain model classes. At startup, it creates a `SYSTEM_ENTITY` object and then parses one or more policy files. Policy files contain one or more creation, assignment or filter rules. Remember that rules themselves are also objects in the system. Thus, the translation controller creates an object for each rule.

At this moment, there is one `SYSTEM_ENTITY` object and an object for each rule. The translation controller then iterates over all available objects and asks them to configure themselves. This configuration process is different for `RULE` and `ENTITY` objects. `RULE` objects check if there are new objects that conform to their selection clause. If there are, they attach themselves to those objects.

The configuration process of an `ENTITY` object starts with checking all attached creation rules. The creation rules are sorted by class name. Remember that classes represent things such as DHCP servers and web servers. For each class name, the intersection of all creation constraints is computed. Creation constraints are constraints on the number of objects of each class name. If the intersection of all creation constraints is empty, an error is generated. Else, the minimum number of objects is created to satisfy all creation rules.

Next, all attached assignment rules are checked and sorted per attribute name. For each attribute, the set of allowable values is computed. If this set is empty, an error is generated. Else, the number of elements in the set is computed. If there is only one element, the attribute's value can be assigned. Else, one value is chosen from the set. The algorithm that chooses one value from a set can be redefined in each class. For example, the algorithm for choosing a valid port on a web server will have to take into account ports chosen by other services on the same device. The algorithm for choosing a valid IPv4 address from a set will have to take into account the network address of its subnet and addresses already assigned to other devices on the same subnet.

After all objects have been asked to configure themselves for the first time, the whole process is repeated. In practice, `SYSTEM_ENTITY` will create a number of `DEVICE` objects based on its attached creation rules. In the next run of the configuration process, `DEVICE` objects will create other objects

representing services like DHCP servers and web servers.

The configuration process continues until all existing objects reach a stable state. A stable state for an object is defined as follows: all rules attached to the object are satisfied. A class can extend the definition of a stable state. In the `RULE` classes, for example, the definition of stable state is extended with the requirement that a rule must be attached to all objects satisfying its selection clause. For a web server class, the definition can be extended with the requirement that the port attribute must have a value, even if no rules exist that set the port attribute. Determining values for attributes for which no rule exist is done by calling an extra method after the configuration process of each object. By default this method contains nothing, but objects can redefine it. For example, the web server class can define this method to set the port attribute to 80 if no rules exist for this attribute.

It is possible that a stable state is never reached. First, a class definition can be erroneous. The specification of what is a stable state can be ill-defined. The extra method that can be defined in each class for additional configuration can also contain errors that prevent objects from the class (or other objects) to reach a stable state. Second, because of the complex (multiple) inheritance relationships that can exist between classes it is possible that the creation rules are never satisfied.

The enforcement of filter rules is done when objects execute methods on each other. Before executing a method, the attached filter rules are checked. If a block policy exists, the execution is not allowed.

Configuration Deployment

When the translation controller notices that all objects have reached a stable state, the deployment process is started. The goal of the deployment process is to generate configuration files from the created objects and deploy these files on all managed devices.

The first phase is to output an XML-based representation of the in-memory objects. This is done by asking the `SYSTEM_ENTITY` object to output its configuration. The `SYSTEM_ENTITY` object asks all its children (which are `DEVICE` objects) to output their configuration. The `DEVICE` objects in turn, ask their children to output their configuration and so on. The result is that, for each `DEVICE`, a tree-structured XML profile is created. This profile consists of simple attribute-value assignments for all attributes and queries of an object. The format of this XML representation defined in Anderson's and Smith's LISA 2005 paper [8].

Next, the XML profiles are used as input for a template engine which generates configuration files and associated configuration instructions. Except configuration files themselves, everything that can be changed in a system is defined as a configuration instruction. Examples are: settings permissions and ownerships of files, installing packages, restarting software services

and creating links. The format of configuration instruction is XML-based and is derived from the internal XML format that Bcfg2 [18, 20, 19] uses. The grammar of the format can be found on <http://purl.org/podim/devel>. The configuration instructions are then translated to the languages used by one of the deployment backends. The prototype allows multiple deployment backends to be used. For example, it is possible to translate configuration instructions to Cfengine [12, 10, 11, 14], Bcfg2 [18, 20, 19] or Lcfg [7, 3, 6] specifications. It is also possible to add additional deployment backends.

Case Study

To validate our system, we use the IPv4 addressing policies for the Computer Science Department of the K. U. Leuven (CSNet). CSNet has a total of 600 machines in about 20 subnets. The 134.58.39.0-134.58.47.255 block of addresses is assigned to CSNet. CSNet has two connections to the university-wide network. The main connection is a subnet that contains, besides the external router for CSNet, switches from other departments and a router that connects to the main K. U. Leuven backbone. One lab is connected to the private network of the K. U. Leuven. We want to assign static addresses to all network interfaces. Some subnets need private addresses. Private addresses are used by the lab networks and the network of the departmental administration, since this is a Windows network which is safer behind a NAT device.

Besides classes for modeling devices and network interfaces, we need a class to model a static IPv4 address configuration. This class is shown in Listing 13. The class contains a reference to a network interface (line 6), the value for its address (line 9) and its network (line 12). The class also defines a query that returns the netmask (lines 17-20).

```

01 class
02     NETWORK_IPV4_STATIC_ADDRESS
04 feature -- Attributes
06     interface: NETWORK_INTERFACE
07         -- attached interface
09     address: IPV4_ADDRESS
10         -- IPv4 address
12     network: IPV4_NETWORK
13         -- subnet configuration
15 feature -- Queries
17     netmask: INTEGER is
18         do
19             Result := network.netmask
20         end
22 end

```

Listing 13: Class definition of a static IPv4 address.

The IPv4 addressing policies for CSNet are described in Listing 14. Because of space limitations, we omitted the creation of device and network interface

objects, representing the hardware configuration of our infrastructure. Notice that, except for a few corner cases (the networks providing external access), all devices are managed with the first three constraint rules: one creation constraint rule that creates static IPv4 address configuration and two rules for configuring the private and public address space. In plain English, the rules in Listing 14 read as follows.

1. **Rule on lines 3-8:** All network interfaces must have one static IPv4 address configured, except for the interface of “jasje” on the external access subnet (KULEUVENNET). “Jasje” is our network sniffer.
2. **Rule on lines 10-14:** All network interfaces that must be reachable from the Internet must have an IPv4 address in the range 134.58.39.0 - 134.58.47.255.
3. **Rule on lines 16-20:** All network interfaces on private subnets must have an IPv4 address in the range 192.168.0.0 - 195.168.255.255.
4. **Rule on lines 22-26:** The network interfaces on the PC_KLAS subnet must have an IPv4 address on the 10.2.15.0/24 subnet.
5. **Rule on lines 28-32:** The access switch on the PC_KLAS subnet must have the 10.2.15.254 address.
6. **Rule on lines 34-38:** All interfaces in the external access subnet – KULEUVENNET – must have an IPv4 address on the 134.58.254.64/29 subnet.
7. **Rule on lines 40-45:** The gateway of the external access subnet must have the 134.58.254.70 address.

As discussed previously, the translation controller reads the policy rules and tries to find a stable state. If the latter succeeds, configuration files are generated by the template engine. Based on the operating system of a device, a template file is chosen. This template then generates configuration files. For example, for OpenBSD devices, /etc/hostname.xxx files are generated. For Debian GNU/Linux devices, /etc/network/interfaces are generated. On Cisco routers and switches, one global configuration file is generated. Depending on the mechanics of the chosen deployment engine (Cfengine, Bcfg2, Lcfg, ...), configuration files are then transported to and deployed on a device.

Related Work

Related work of PoDIM’s high-level configuration language includes configuration management tools. We also discuss how generic policy languages and a model finder are related to the problem PoDIM is trying to solve. We end this discussion with a characterization of application deployment frameworks.

Configuration Management Tools

Bcfg2 [18, 20, 19], Cfengine [12, 10, 11, 14], LCFG [7, 3, 6] and Puppet [26, 27] are the most cited

configuration management tools. As discussed previously, these tools can be used as a deployment backend for PoDIM. Bcfg2 and Cfengine are in the first place deployment engines. LCFG and Puppet include capabilities for modeling dependencies between configurations.

Other related work in the context of configuration management includes the work of Couch on closures [16]. Closures are defined as functional units that can accept commands from the user or other closures. Their internal mechanics are hidden. The classes from PoDIM's domain model can be seen as closures. Classes define commands that change the behaviour of their objects, but can also have queries and attributes.

Policy Languages

PoDIM separates the domain model and the policy specification language. Many other policy languages use this separation. It allows for reuse of both the policy specification language and domain model in

different contexts. The PCIM [33, 32] (Policy Core Information Model) and CIM (Common Information Model) [15] initiatives define a generic model for representing policy specifications on one side and a set of domain classes on the other side. The generic model defines policy rules in a Condition-Action format. The domain model includes definitions for common network functionalities such as routing protocols, network configurations and IPsec configurations. The domain model itself is object-oriented and models relations between classes. The CIM domain model provides a valuable repository of existing domain knowledge, modeled as object-oriented classes. The CIM model is very similar to the PoDIM domain model. However, it has no support for specifying fine-grained dependencies. PoDIM uses Eiffel invariants for this. PCIM/CIM also does not support constraint handling.

Other frequently cited policy languages such as Ponder [17] and JRules [23] also offer an extensible domain model. The domain model of these languages

```

1  -- IPv4 Addressing
3  creation
4    -- Each interface has 1 IPv4 address, except "jasje"
5    [1-1] NETWORK_IPV4_STATIC_ADDRESS
6    select NETWORK_INTERFACE
7    where NETWORK_INTERFACE.device.name /= "jasje" and
8          NETWORK_INTERFACE.labels.has("KULEUVENNET")
10 assignment constraint
11   -- Public address space
12   address [!!IPV4_ADDRESS.make("134.58.39.0")::!!IPV4_ADDRESS.make("134.58.47.255")]
13   select NETWORK_IPV4_STATIC_ADDRESS
14   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PUBLIC_SUBNET")
16 assignment constraint
17   -- Private address space
18   address [!!IPV4_ADDRESS.make("192.168.0.0")::!!IPV4_ADDRESS.make("192.168.255.255")]
19   select NETWORK_IPV4_STATIC_ADDRESS
20   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PRIVATE_SUBNET")
22 assignment
23   -- PC_KLAS IPv4 Address range
24   network !!IPV4_NETWORK.make("10.2.15.0",24)
25   select NETWORK_IPV4_STATIC_ADDRESS
26   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PC_KLAS")
28 assignment
29   -- PC_KLAS external router
30   address !!IPV4_ADDRESS.make("10.2.15.254")
31   select NETWORK_IPV4_STATIC_ADDRESS
32   where NETWORK_IPV4_STATIC_ADDRESS.interface.device.name = "lswitch-cw"
34 assignment
35   -- KULEUVENNET external access
36   subnet !!IPV4_NETWORK.make("134.58.254.64",29)
37   select NETWORK_IPV4_STATIC_ADDRESS
38   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("KULEUVENNET")
40 assignment
41   -- Gateway configuration for KULEUVENNET
42   address !!IPV4_ADDRESS.make("134.58.254.70")
43   select NETWORK_IPV4_STATIC_ADDRESS
44   where NETWORK_IPV4_STATIC_ADDRESS.interface.device.name = "default-gateway" and
45         NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("KULEUVENNET")

```

Listing 14: Policy Specification for the network configuration of K. U. Leuven's CS department.

is object-oriented, as is the case with the PoDIM domain model. Policy rules are Event-Condition-Action based in these languages. The action that can be executed is an arbitrary operation of the domain model. Notice that differs with our approach. Our approach is less expressive in the sense that we do not allow the execution of arbitrary operations. However, we do allow to model constraints on attributes, which is something that is not supported by the current generation of policy languages.

Model Finding

In [35] a model finding approach is proposed for configuration management based on Alloy [24, 25, 1]. This approach is based on creating a model for an infrastructure based on first-order logic. Based on a number of inputs (such as the number of devices and network interfaces) an outcome is constructed that satisfies the model. The advantage of using a tool such as Alloy is that it allows very advanced reasoning over a configuration. The same model can be used to generate and validate configurations. The limitations are that constraints, as we discussed them in this paper, can not be modeled. It is also difficult to set specific attributes of “things”. For example, it is difficult to set a human readable name for every device.

Application Deployment Frameworks

Application deployment frameworks like SmartFrog [29, 28], Spring [38] and JBoss Microcontainer [22] manage applications directly by tuning their parameters. Notice the difference with PoDIM: classes directly control real world things, while PoDIM classes represent real world things that are deployed when a stable state is reached. Because of this difference, application deployment frameworks listed above do not provide constraint resolution of the kind that PoDIM uses.

Future Work

There are various areas where PoDIM could be improved. We already mentioned the stable state problem: in some cases, it is impossible to reach a stable state and, as a consequence, generate a valid configuration. It is also possible that, on different runs of the translation controller, different configuration files are generated for the same input rule sets. This is because of the possibility that classes define random choices for choosing a value from a constraint set. Including information about previous runs could solve this problem.

A dry-run or analysis mode would also be useful. Currently, the prototype supports the actual translation and deployment process. Checking the validity of a rule set requires a simulation modus. Ponder, for example, uses Event Calculus [9] to do this.

We have not yet gathered data on the scalability of our prototype. In its current implementation, the translation from rules to configuration files is done on

one central component. We are currently working on a decentralized version of the translation controller. In the decentralized version, each device can be made responsible for generating its own configuration and will need to communicate with other devices to find a globally valid state.

Other areas for improvement deal with usability.

- Extending the domain model requires knowledge of Eiffel. In many cases, a less expressive (and easier) notation suffices for creating new classes in the domain model. A program can then translate classes to the Eiffel notation.
- There is no support to track configurations throughout the translation process. Therefore, it is impossible to know what rules influence the generation of a configuration files or what changes in a configuration file are caused by a change in one of the rules. Both would be useful for debugging rules. In general, the translation controller must be able to explain why a configuration is generated.
- The rule language contains no structuring mechanisms. Working with large policy sets becomes cumbersome. Existing preprocessor systems can solve part of this problem, but specifying meta-rules will stay inconvenient. Instead of looking at PoDIM's rule language as a user interface, it is better to see it as a target format for more advanced interfaces which could be, depending on the case text-based, command-line programs, graphical user interfaces, web interfaces, and so on.

Conclusion

PoDIM tries to address the complexity of configuration management by abstracting from system variety and providing mechanisms for specifying cross machine constraints. The PoDIM language consists of a rule language and an extensible domain model. The current prototype translates high-level rules in low-level configuration files and subsequently uses existing configuration management tools to deploy the generated configuration on all managed devices. We believe that PoDIM provides an advancement of the state of the art. It provides a higher-level specification compared to what is currently available, which includes cross machine constraints and abstracts away systems variety.

Acknowledgments

This research has been supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (<http://www.iwt.be>). We would like to thank Sara Vermeylen and Nico Janssens for proofreading this paper. Thanks also to Ed Smith for his excellent work on shepherding this paper, and Paul Anderson for discussing configuration management ideas.

Author Biographies

Thomas Delaet is a Ph.D. student at the computer science department of the Katholieke Universiteit Leuven. His research is funded by the IWT, the Flemish institute for innovation in science and technology.

Wouter Joosen is a full professor in Distributed Systems at the Katholieke Universiteit Leuven, Belgium. He has been a professor in software engineering at Odense University, Denmark, from 1997 to 2001. Wouter's research interests include the development, deployment and management of distributed and secure software systems.

Bibliography

- [1] *The Alloy Analyzer*, <http://alloy.mit.edu>.
- [2] American National Standards Institute, *ANSI X3.135-1992: Information Systems – Database* 1430 Broadway, New York, 1989.
- [3] Anderson, Paul, *LCFG Homepage*, <http://www.lcfg.org>.
- [4] Anderson, Paul, *Short Topics in system Administration 14: System Configuration*, USENIX Association, Berkeley, CA, 2006.
- [5] Anderson, Paul and Alva Couch, "What Is This Thing Called "System Configuration?" LISA Invited Talk, November, 2004.
- [6] Anderson, Paul, and Alastair Scobie, "Large Scale Linux Configuration with LCFG," *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, October 10-14, pages 363-372, USENIX, Berkeley, CA, 2000.
- [7] Anderson, Paul and Alastair Scobie, "LCFG – the Next Generation," *UKUUG Winter Conference*, UKUUG, 2002.
- [8] Anderson, Paul and Edmund Smith, "Configuration Tools: Working Together," *Proceedings of the Large Installations Systems Administration (LISA) Conference*, pages 31-38, USENIX Association, Berkeley, CA, December 2005.
- [9] Bandara, A. K., E. C. Lupu, J. Moffett, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," *Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks*, 2003.
- [10] Burgess, M., *Cfengine WWW site*, 1993, <http://www.iu.hio.no/cfengine>.
- [11] Burgess, M., *GNU cfengine*, Free Software Foundation, Boston, Massachusetts, 1994.
- [12] Burgess, M., "A Site Configuration Engine," *Computing Systems*, MIT Press: Cambridge, MA, Vol. 8, p. 309, 1995.
- [13] Burgess, M., "Needles in the Cray Stack: The Myth of Computer Control," *USENIX ;login:*, Vol. 26, Num. 2, pp. 30-36, 2001.
- [14] Burgess, Mark, "Recent Developments in Cfengine," *Unix.nl Conference Proceedings*, 2001.
- [15] *Common Information Model (CIM) Standards*, <http://www.dmtf.org/standards/cim/>.
- [16] Couch, A., J. Hart, E. G. Idhaw, and D. Kallas, "Seeking Closure in an Open World: A Behavioural Agent Approach to Configuration Management," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, CA, p. 129, 2003.
- [17] Damianou, Nicodemos C., *A Policy Framework for Management of Distributed Systems*, Ph.D. thesis, University of London, Department of Computing, 2002.
- [18] Desai, Narayan, Rick Bradshaw, and Joey Hagedorn, *Bcfg2 Trac Homepage*, <http://trac.mcs.anl.gov/projects/bcfg2>.
- [19] Desai, Narayan, Rick Bradshaw, and Joey Hagedorn, *System Management Methodologies with Bcfg2*, ;login: *The USENIX Association Newsletter*, Vol. 31, Num. 1, February 2006.
- [20] Desai, Narayan, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Rémy Evard, Cory Lueninghoener, Ti Leggett, John-Paul Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey, "A Case Study in Configuration Management Tool Deployment," *Proceedings of the Large Installations Systems Administration (LISA) Conference*, pp. 39-46, USENIX Association, Berkeley, CA, December, 2005.
- [21] Evard, R., "An Analysis of UNIX System Configuration," *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX Association, Berkeley, CA, p. 179, 1997.
- [22] JBoss Group, *Jboss Microcontainer*, <http://www.jboss.com/products/jbossmc>.
- [23] ILOG, *Jrules: Technical White Paper (Version 4.0)*, 2002, <http://www.ilog.com/products/jrules/>.
- [24] Jackson, Daniel, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology*, Vol. 11, Num. 2, pp. 256-290, 2002.
- [25] Jackson, Daniel, *Software Abstractions: Logic, Language and Analysis*, The MIT Press, 2006.
- [26] Kanies, Luke, *Puppet*, <http://reductivelabs.com/projects/puppet/>.
- [27] Kanies, Luke, *Puppet: Next-Generation Configuration Management*, ;login: *The USENIX Association Newsletter*, Vol. 31, Num. 1, February, 2006.
- [28] Hewlett Packard Laboratories, *The smartfrog Reference Manual*, 2007.
- [29] Low, Colin and Julio Guijarro, *A smartfrog Tutorial*, Technical Report, Hewlett Packard Laboratories, 2004.
- [30] Meyer, B., *Eiffel, the Language*. Prentice Hall, 1992.
- [31] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice-Hall, Second Edition, 1997.

- [32] Moore, B., *Policy Core Information Model (PCIM) Extensions, RFC 3460 (Proposed Standard)*, January, 2003.
- [33] Moore, B., E. Ellesson, J. Strassner, and A. West-
erinen, *Policy Core Information Model – Version
1 Specification, RFC 3060 (Proposed Standard)*;
Updated by RFC 3460, February, 2001.
- [34] Narain, Sanjai, *Towards a Foundation for Build-
ing Distributed Systems via Configuration*, 2004,
[http://www.argreenhouse.com/papers/narain/Service-
Grammar-Web-Version.pdf](http://www.argreenhouse.com/papers/narain/Service-Grammar-Web-Version.pdf).
- [35] Narain, Sanjai, “Network Configuration Manage-
ment via Model Finding,” *LISA'05: Proceedings
of the 19th Conference on Large Installation Sys-
tem Administration Conference*, p. 15, USENIX
Association, Berkeley, CA, 2005.
- [36] Oppenheimer, D., “The Importance of Understand-
ing Distributed System Configuration,” *Proceed-
ings of the 2003 Conference on Human Factors
in Computer Systems workshop*, April, 2003.
- [37] Patterson, D. A., “A Simple Way to Estimate the
Cost of Downtime,” *Proceedings of the Sixteenth
Systems Administration Conference (LISA'02)*, pp.
185-188, USENIX Association, Berkeley, CA,
2002.
- [38] *Spring Framework*, <http://www.springframework.org>.
- [39] Strassner, John, “Policy management challenges
for the future,” Policy 2005 Keynote, 2005.
- [40] Walden, Kim and Jean-Marc Nerson, *Seamless
Object-Oriented Software Architecture*, Prentice-
Hall, 1994.

Network Patterns in Cfengine and Scalable Data Aggregation

Mark Burgess and Matthew Disney – Oslo University College
Rolf Stadler – KTH Royal Institute of Technology, Stockholm

ABSTRACT

Network patterns are based on generic algorithms that execute on tree-based overlays. A set of such patterns has been developed at KTH to support distributed monitoring in networks with non-trivial topologies. We consider the use of this approach in logical peer networks in cfengine as a way of scaling aggregation of data to large organizations. Use of ‘deep’ network structures can lead to temporal anomalies. We show how to minimize temporal fragmentation during data aggregation by using time offsets and what effect these choices might have on power consumption. We offer proof of concept for this technology to initiate either multicast or inverse multicast pulses through sensor networks.

Introduction

In this paper we consider an approach for scaling data dissemination (e.g., for configuration management) or alternatively for scaling data aggregation (e.g., for monitoring or archiving) by implementing Network Patterns on top of cfengine’s pull-based copy methods. This follows up preliminary work on scaling in [1, 2] and Voluntary Cooperation [3] and is inspired by work on the Generic Aggregation Protocol (GAP) described in [4, 5, 6].

Consider the sharing of load in a multicast process by handing off parts of a task to decentralized processing. For example, in a distributed backup scheme, one could imagine assigning responsibilities such that local nodes collected and compressed their own data before passing them from the leaves of a tree to their parent node; the parent would then aggregate data from all of its children and adds its own data, and so on up the tree to a final repository. By introducing several tree levels one reduces the total computational burden on the final host. Such a strategy could be useful in either a fixed infrastructure network (where nodes have limited computational power) and especially in battery powered processors such as wireless ad hoc devices, sensor networks, and so on.

Network Patterns are based on generic, distributed algorithms that execute on spanning trees, designed to collate information from a topologically constrained network, such as a fixed routing infrastructure or ad-hoc substrate. They employ the basic structures used in routing and switching, like spanning trees, and can adapt to node or link failures [4, 5, 6]. We shall consider only aggregation algorithms here, where aggregates of local variables across a domain of local devices are computed using functions, such as sum, max, or average.

The overlay networks are usually created under some basic physical constraints such as geography,

physical network design, allowed access, or even by wireless power limitations in an ad hoc network. In other words, certain branches and levels in the tree could be forced into the final topology by physical circumstances, hence one could not merely choose the simplest star topology for the task, even if it were not an unacceptable burden on the single bottleneck. However, we can also ask whether it makes sense to build such structures even where there are no constraints, such as local area networks with underlying star topology. There are valid resource sharing reasons for doing this in system administration, especially where resources are limited.

Network patterns allow a kind of load balancing, but they are different from the kind of service balancer which one might use on a web server: a traditional load-sharing dispatcher acts like a switch, taking a single input stream and offloading it to a separate queue: in a network pattern data are sent to all branches, like a “smart” multi-port repeater or amplifier/aggregator.

Inter-Domain Management and Voluntary Cooperation

A subject that is increasingly discussed in today’s world of cooperative outsourcing is the issue of *inter-domain management*. In the extreme case, each node in a network is in its own administrative domain (this is approximately true for border routers, for example, as well as hand-held devices). Inter-domain management involves many issues that are often ignored in discussions of system administration. For example, we do not typically have privileged access to all of the devices we communicate with. The concept of Voluntary Cooperation was introduced to discuss “minimal trust” interactions with autonomous domains [3].

Even a wireless ad hoc network of personal electronic devices (or a military network deployed in the

field) could be formed from many devices with different privileges and privacy policies. Traditional models of centralized control do not begin to address these issues.

Monitoring (data collection) from a network of sensors (either in a fixed infrastructure net or in a wireless environment) is an application that has received a lot of attention. This is because “network management” has traditionally been about watching network traffic data. Even today as vendors advocate the virtues of autonomic computing, network managers still want to watch the automation in progress. Thus the problem of distributed aggregation with unclear domain relationships is still at the heart of network management.

Cfengine is a management system that represents state of the art research on integrating monitoring and reactive (“autonomic”) management of computers. Integrating network patterns into cfengine would allow distributed monitoring and management of a manifestly autonomic system with any chosen degree of centralization or decentralization. Cfengine is designed to be able to work in mobile, partially connected environments. It is an ideal testbed for exploring the usefulness of patterns in host based system administration. Moreover, eventually it is expected that cfengine will be able to manage routers and switches for which patterns were originally envisaged.

Network Patterns are not generic routing or switching structures, although they share similarities. They are designed to execute any computation whose data can be represented on the underlying graph. This typically involves aggregation, dissemination, maximization or minimization etc. Here we use them only for the simplest aggregation of data from every node

in a network to an arbitrary but central place. They are therefore used to initiate either multicast or inverse multicast pulses through sensor networks.

Any collection of “sensor devices” that can run on a GNU/Linux platform could use cfengine in the way we demonstrate here, and this accounts for an ever increasing number of devices available today. One application is for collecting and correlating data from around a network from cfengine’s own sensor component cfenvd. Cfengine’s investment in methods of *voluntary cooperation* means that one need not give away privileges in order to implement patterns, hence risking or sacrificing security. This makes monitoring of large and fragmented organizations an easier process to swallow for security officers (the alternative being to open firewalls to unspecified network pushes). Increasingly companies are outsourcing their systems into different formal domains with their own policies and barriers. The fact that one can make patterns work with voluntary cooperation is therefore itself a valuable proof of concept.

A natural application for this kind of process is for monitoring grid systems. These are systems that are often geographically distributed and already form part of some organized structure. Patterns at the level of host based monitoring would allow grid administrators to view the performance characteristics of the component systems or even aggregate results from them with controllable accuracy.

There are various other applications for data aggregation to a point. Another one is to perform a distributed backup, collecting and compressing data as they propagate up the tree. This would offload the

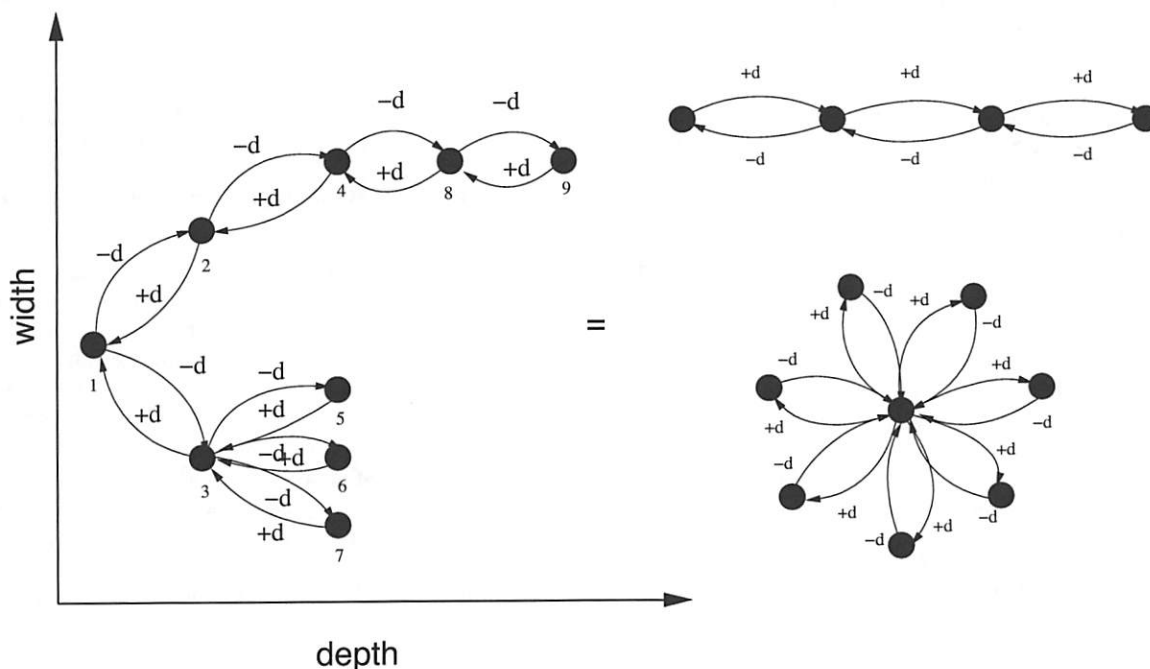


Figure 1: Depth and width in network patterns formed from promises.

burden of performing the compression, and data could be encrypted with local keys before compression. We shall not elaborate on these applications here, but simply present these tests as proof of the concept.

Some Patterns

The patterns discussed here are dissemination and aggregation algorithms that bridge the worlds of centralized monitoring and fully distributed monitoring. They are built from “component” pieces that represent the extreme cases of any network structure: *chain* (for maximum depth) and the *star* topology (for maximum width), see Figure 1.

Trees are structures that bridge these two extremes. We can characterize patterns by their depth and breadth. Note that a chain is also geometrically a half-ring, so it gives us a basic model for ring-topologies also.

Here we consider only two patterns: Echo and GAP and consider how these can be implemented in cfengine using existing context awareness within the system.

Echo

The simplest example of a network pattern is the *echo* pattern [7, 8]. During its execution, echo creates

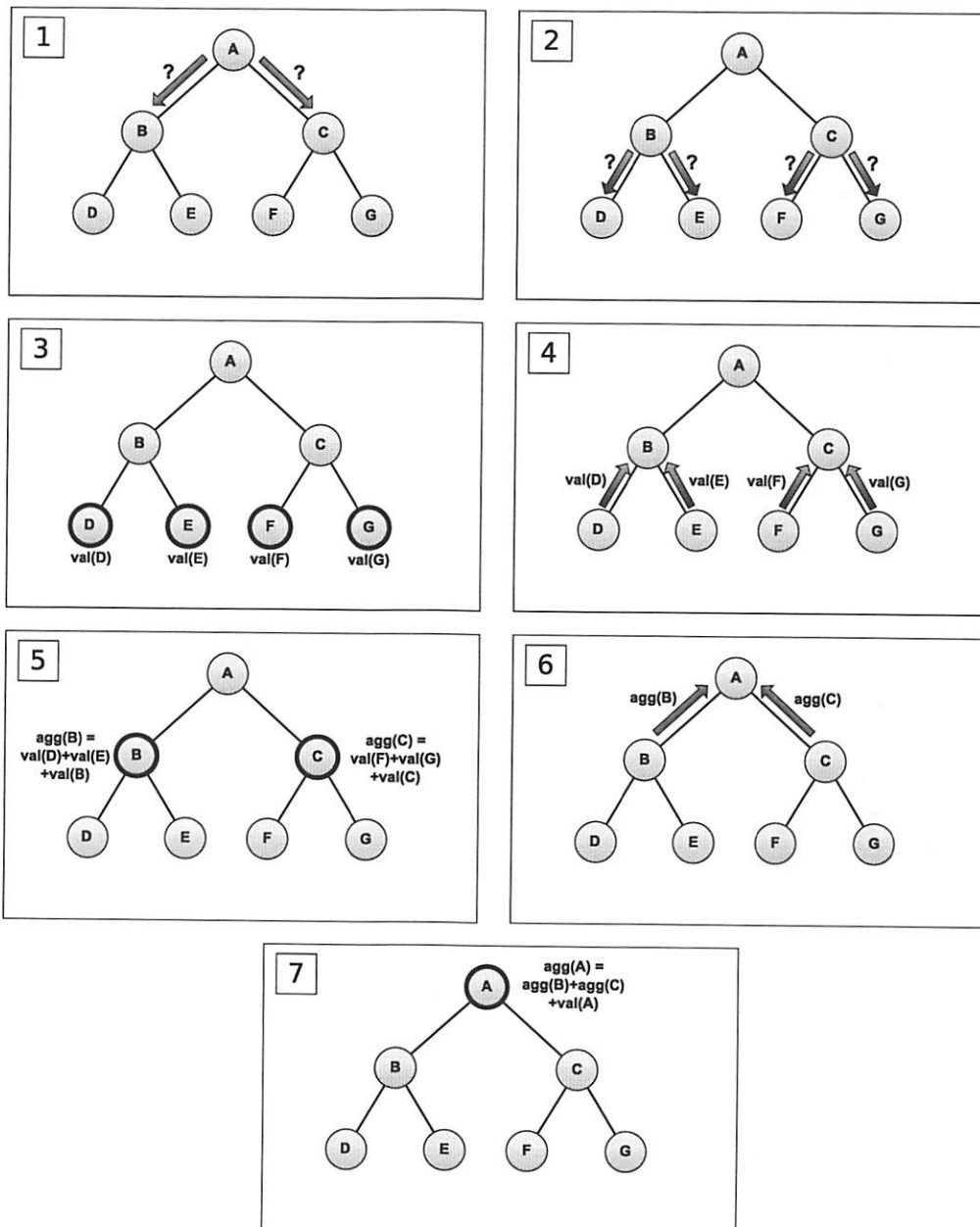


Figure 2: The expansion and contraction steps in the echo pattern. The pattern is a sequence of “pulls” initiated by “pushed” signals.

a spanning tree topology, with root of the tree chosen by an administrator. The pattern has two phases of communication: *expansion* and *contraction* (see Figure 3). During the expansion phase, the root node issues a query to its children. Each node in the tree repeats this process. The contraction begins as the query reaches a leaf node. The leaf node answers the query, sending its response to its parent in the tree. The parent receives the response of its children, aggregates or calculates information for the query to the fullest extent possible, and then sends a single aggregate answer to its own parent. This process is repeated recursively, until the root node is reached, which aggregates the messages from its children. The tree topology provides for parallelized execution, while the aggregation of query responses during contraction reduces the amount of traffic that would otherwise be necessary. The echo pattern therefore forms a wave, spreading out from the root to the edge of the network and back, collecting data as it progresses. Echo is intrinsically a “push” protocol, and is easily understood as a recursive descent parser.

GAP

Similar to Echo, the Generic Aggregation Protocol (GAP), creates a spanning tree along which communication and computation takes place. Unlike echo however, data in GAP are passed from the leaves of the tree towards the root, whenever the local variable in one of the nodes changes. Updates to monitored aggregates can thus be initiated by any node, not only by the root node. Thus GAP, once initialized, responds to local events rather than initiating measurements from a central observer.

GAP is an asynchronous distributed protocol that builds and maintains a Breadth First Search (BFS) or a spanning tree over which aggregates are computed incrementally and continuously. The tree is maintained in a similar way to the algorithm underlying the 802.1d Spanning Tree Protocol (STP). In GAP, each node maintains a table of its peers and especially its nearest neighbours along with an estimation of the nodes’ aggregate values. GAP is event driven, in the sense that each update from a leaf node triggers a cascade of events through the tree branches, updating the local aggregates as it goes. Update events can be triggered by changes in topology, loss of a node, a timer, etc.

The advantage of GAP over echo is that there is no “push phase” required to initiate a reading of the values from the network. As each change occurs in the network, new values can be percolated back to the centralized root node initiating an update only in those tree nodes that are in the path to root. This avoids the need for much unnecessary traffic and computation during updates.

The Topology Manager

A key feature of the patterns above is the algorithm by which the topology of the spanning tree is

decided. The GAP algorithm incorporates the topology adjustment mechanism into the GAP aggregation algorithm, by using nearest neighbour communications, hence combining these features into a robust protocol. However, they can be separated also. The GoCast algorithm finds such a spanning tree, for example. At this stage of the work we shall not attempt to encode automated topology management, as this requires additional subsystems. Rather we consider how patterns can be used at the logical level for distributed load balancing, using existing mechanisms within cfengine. We note however that cfengine has implemented peer neighbour management functions for some time in the form of the

```
SelectPartitionNeighbours
SelectPartitionLeader
```

functions. These functions take a flat list of all known hosts and partition this list into clusters of a specified size. Each cluster is assigned an identified leader which can be used to single out a root or responsible node for each group, and in this way any host can autonomously be made aware of its nearest neighbour topology based only on the shared information of the flat list. These functions, or functions like them could in principle be used to provide an implementation of GAP topology management in future, for automatically adaptation for fault tolerance. However, we shall not pursue the details of the topology here, since it turns out that the implementation of patterns throws up a number of issues that are more fundamental.

The problem of building soft-overlays for computational load sharing is slightly different to the problem of finding a spanning tree through a physically redundant topology however. In principle, any kind of overlay could be built in software, but physical constraints can limit the potential optimizations. What we find interesting in a cfengine environment is that we must deal with a combination of these issues. If cfengine is used in a simple star network, any kind of overlay can be built. However, if it is used for inter-domain management, or between zones with different administrative regimes, then these amount to essentially physical constraints.

Cfengine Principles and Patterns

By implementing network patterns in cfengine we hope to achieve two things: i) an efficient way of aggregating data for centralized analysis and decision-making, and ii) open for the possible load sharing optimizations that are possible with patterns. An obvious goal for centralized decision-making would be to use this to build an “autonomic nervous system” from cfengine’s autonomous agents so that centralized monitoring and decision-making can be added to its local stimulus-response approach to management. Although many configuration management schemes boast “centralization”, this can often be seen as a weakness, as it is a clear limitation on scalability, and such systems usually only disseminate data from a centralized source: we

are advocating stimulus-response in a distributed system, something like a central nervous system. While individual machines work autonomously, we collect, process and return data to the nodes on a continuous basis.

The desired model is not without its own challenges however: cfengine maintains strong principles of autonomy that are largely responsible for its record of security and reliability. The challenge is to implement aggregation/dissemination patterns without sacrificing those strong principles.

A cfengine host is, by default, a completely autonomous entity with no obligations towards other agents in a physical network. Every node is therefore individual and is not part of a pattern a priori. Leaf nodes cannot initiate a push of new data in response to events, because the parent node does not accept data from any outside source, unless it explicitly pulls the data itself. To use patterns as a form of inter-peer collaboration, we must encode them as policy rules that are compatible with cfengine's pull-only principle of communication. There are several questions to be answered about this:

- Is the underlying physical network topology important in building a logical load sharing topology?
- How will the topology be decided?
- How will the topology respond to the failure of nodes?

We shall not be able answer all of these questions, but we present the basic approach to building GAP-like patterns using cfengine's internal mechanisms, and provide tools for readers to experiment on their own.

Periodic Execution

Cfengine is normally used for regular (periodically) scheduled maintenance sweeps, yet the traditional idea of a network probe is to ask a question and get back the answer on demand (as with probes like ping and traceroute). The Echo pattern is a "push-me pull-you" strategy for connecting to all elements in a managed network and transmitting or collecting data: a kind of broadcast ping. The principal advantage of this kind of approach is that the timing of the distributed process is event driven. It does not require an elaborate clock synchronization and timed firing to coordinate the distributed execution, since the interactions are themselves synchronous. However, it is inherently fragile as it involves the privilege to push and collect through a chain of dependencies. If the top node loses communications with its children, none of the network operations will be executed. A better approach would be allow all nodes in the network to operate *autonomously* and have them cooperate when they are able.

A typical cfengine approach to the problem to execute the distributed agents periodically (with period P ,

anything from a few minutes to an hour). Neighbouring cfagents could download from their children servers to aggregate the results, but now the timing plays a role. Since there is no push possibility to coordinate the operations, the process is fragile to time coordination [9]. There are two issues: i) clock synchronization and ii) clock schedule for ensuring the data are updated in time before the data values are pulled downstream. If either of these requirements is not met, data that are pulled will be out of date and will not give an accurate representation of the true values.

So what happens if the nodes are not properly synchronized? Since cfengine operates autonomously and its copying is fault tolerant, a missed update could simply be captured at a later time. This might not seem like a problem, unless one begins to measure the spread of times in the "current" data. The situation is somewhat analogous to asking post office branches to report to their head office on how many customers they have each day using their own postal delivery. At any given regular delivery, the letters that arrive at the central office have a variety of postmarks. Some of them are delivered on the same day, and some of them take perhaps a week to deliver. Thus updates might arrive eventually, but how shall we understand the results that arrive? Do we group letters by their postmarks and only combine results that were originated on the same date? Or do we ignore the post-marks and combine data that were received on the same date? In the first case, we might have to wait a long time for the data, but we are certain of what we are seeing. In the latter case, the result is available quickly but the meaning of the data is in question.

Each hop in a chain of delivery adds new possibility for delay. If the mail does not arrive before one post office sends its own delivery, the incoming mail will have to wait a whole day for the next delivery (a whole scheduling period P). A single failure could not bring down the entire system, but it could skew the impression received at the central monitoring station. It is therefore advantageous, if not imperative, to develop patterns that do not have this strong dependency feature.

To avoid the dependency and delay problem, we based our work on the assumption of time synchronization. As we shall see, even this is susceptible to noise. Apart from a proof-of-concept implementation, we did not pursue the echo pattern for this reason (in spite of its ready comprehensibility) and instead were inspired by the Generic Aggregation Protocol (GAP) approach. For GAP we shall not attempt a complete implementation, but rather emulate its operation as a first step to making progress. GAP includes an algorithm for automatic renegotiation of the structure. This has several implications which require some soul searching when implementing in cfengine. Further research by KTH based on the cfengine experience can also help to adapt the GAP algorithm for pull-based scenarios.

Promise Agreements and Voluntary Cooperation

The notion of promises was introduced as a way of modeling networks of agents cooperating in an ad hoc fashion. Cfengine can be viewed as a reference implementation of the abstract promise-theoretic scenario. Promise theory was introduced precisely as a modeling framework that could describe cfengine, where others could not.

Promise theory is a high level graphical description of constrained behaviour in which ensembles of agents document the behaviours they promise to exhibit. Agents in promise theory are truly autonomous, i.e., they decide their own behaviour, cannot be forced into behaviour externally but can voluntarily cooperate with one another [10]. A promise is a directed edge

$$A_1 \xrightarrow{b} A_2 \quad (1)$$

that consists of a promiser A_1 (sender), a promisee A_2 (recipient) and a promise body b , which describes the nature of the promise. Promises made by agents fall into two basic categories, promises to provide something or offer a behaviour b (written $A_1 \xrightarrow{+b} A_2$, and promises to accept something or make use of another's promise of behaviour b (written $A_2 \xrightarrow{-b} A_1$). A successful transfer of the promised exchange involves both of these promises, as an agent can freely decline to be informed of the other's behaviour or receive the service.

The essential assumption of promise theory is that all nodes are independent agents, with only private knowledge (e.g., of time). No node can be forced to promise anything or behave in any way by an outside agent. Moreover, there are no common standards of knowledge (such as knowing the time of day) without explicit promises being made to yield this information from a source. This viewpoint fits nicely with our view of collection of distributed information for measurement purposes.

We shall consider the following promise designations: $+d$ server provides data, $-d$ client receives/uses data, $+a$ branch node aggregates data, $+t$ server provides time/clock, and $-t$ client uses time/clock. Although we speak mainly of network nodes below, it will be understood that each node is modeled as an "agent" in promise theory parlance.

Promise theory allows us to see the relationship between network patterns and policy for autonomous agents. Each arrow in the promise graph attaches to a rule in the policy to either grant access to data or to fetch available data. In this way we can build dissemination processes over graphs using node location data or context sensitivity information.

A common mistake is to think of promises as communication transactions, rather than as abstract behavioural specifiers. A promise says nothing necessarily about the details of what is communicated between agents at a given moment, only that it intends to

behave within the confines of its promise. However, one usually assumes that a promise means a best effort to comply with the announced constraints and that no promise means that nothing will happen. A reliable binding between two hosts requires both a promise to serve and a promise to use the promised service.

$$A_1 \xrightarrow{+b} A_2, A_2 \xrightarrow{-b} A_1 \quad (2)$$

The Echo and GAP patterns are particularly well suited to implementation using voluntary cooperation, because the propagation of data along tree-like pathways does not depend strongly on whether data are pushed or pulled. The main challenge in a voluntary cooperation scenario is for an agent in the graph to know when its child has data waiting. When data are pushed, we essentially send a signal "do it now", and no other time synchronization is required. This becomes more complicated in a pull regime however. Regular polling of a host's servers is an obvious answer to the question of when to download data. If clocks in the network are synchronized correctly we can even ask for data to be copied only if they have been updated since the last copy. However, this requires the extra overhead of time synchronization and it still does not guarantee that data will be ready for collection at a given moment.

This issue becomes most pronounced when one attempts to request regular pollings of data and the time for data to propagate through the network approaches the time interval for the polling. We have discussed this issue in a separate paper [9], but some of the effects can be seen in Figures 5 and 6.

Using Context Awareness for Making Network Patterns

Cfengine agents are aware of location and context through their evaluation of the environment into a set of classes. These classes are then used as Boolean flags to attach policies conditionally to scenarios. This context sensitivity enables a set of distributed promises to be coded into a single document.

A method in cfengine is like a pair of promises, provided it is voluntarily declared by both parties. An MD5 hash is used to verify that the methods are in fact the same.

The first (service) promise identifies the function being performed, as the body $b()$. The class expression $A_1::$ says that this rule applies to the context of agent A_1 , which is the service provider (server host). The $server=A_1$ attribute matches the context expression and, from this, the agent deduces that it is the provider.

methods:

$A_1::$

$b(params) \text{ server}=A_1$

$$A_1 \xrightarrow{+b} A_2, A_2 \xrightarrow{-b} A_1 \quad (3)$$

The second part applied to agent A_2 and has the form:

```

methods:
  A_2::
    b(params) server=A_1

```

This identifies the function being performed and signals to A_2 that it will use the results performed by server A_1 . Since this is not its own identity, this implies that the result is a use-promise.

If we assume that two agents use an identical configuration specification, then a remote procedure call binding can then be written methods:

```

A_1 | A_2::
  b(params) server=A_1

```

The same text either in both contexts and a single link in a logical overlay network is added.

Echo

Cfengine's modus operandi is to "pull" data rather than to push. This is a natural side effect of its philosophy of voluntary cooperation. Push is disallowed, with one exception: we are allowed to send a single invitation to each peer to execute its existing policy using the command `cfrun`. The host is free to disregard this message, but for cooperation purposes it is normal for the peer to respond to such an invitation by executing its policy compliance-checking agent. We can use this mechanism to start an echo avalanche, with a pre-arranged pattern.

The start host executes `cfrun` to a number of "children". Each child then voluntarily executes cfengine, which in turn encapsulates the execution of `cfrun` directed at another set of children, which encapsulates `cfrun` to another set, and so on. Since cfengine aggregates the data from encapsulated processes automatically, it automatically aggregates the entire tree in a synchronized manner. This is the simplest implementation of echo which uses context sensitivity to identify parent-child relationships.

Both serial and parallel star collation can be performed in cfengine echo. The difference is that the parallel star `cfagent.conf` issues an individual `cfrun` command to each client in the background. Additionally, the output from each of those commands is redirected to a file. When all the `cfrun` processes have finished, the output files are concatenated together and printed to the terminal so that the parallel and serial star tests both provide nearly identical terminal output. However, it should be noted that the parallel star approach, involving the use of a separate temporary file for each client, involves a great deal more file input and output operations than serial star.

The echo `cfagent.conf` draws from the same framework used for the parallel star, e.g., executing `cfrun` commands in the background with output redirected to files. In this case, a variable is defined for each host that has children. The variable contains a list of the node's children in the tree. If this variable is defined, `cfrun` is called for each child node. Therefore the tree is statically defined.

The use-promises are encoded as follows as in Figure 3.

```

control:
  actionsequence      = ( shellcommands tidy )
  domain              = ( cftestnet )
  IfElapsed           = ( 1 )
  TrustKeysFrom       = ( 10.0.0 )

node1::
  serve = ( node2:node3:node4 )
node2::
  serve = ( node5:node6:node7 )
node3::
  serve = ( node8:node9:node10 )
node4::
  serve = ( node11:node12:node13 )
node5::
  serve = ( node14:node15:node16 )
node8::
  serve = ( node17:node18:node19 )
node11::
  serve = ( node20 )

classes:
  HasChildren = ( IsDefined(serve) )
shellcommands:
  "/bin/echo $(hostname)"
HasChildren::
  "/usr/local/sbin/cfrun $(serve) \
    2>&1 > /tmp/echorun.$$"
    background=true # parallelize
  "/usr/bin/pgrep cfrun > /dev/null; \
    while [ $? = 0 ]; \
    do pgrep cfrun > /dev/null; done"
  "/bin/cat /tmp/echorun.*"

tidy:
  HasChildren::
    /tmp pattern=echorun.* age=0

```

Figure 3: The only kind of push structure that can be implemented in cfengine is the echo pattern, using nested `cfrun` commands. These must be authorized in advance.

Promise Chains (Forwarding)

Two implementations of chains are shown in Figures 7 and 8 for readers to try. Conventional wisdom suggests that tree depth corresponds directly to latency in terms of end-to-end communication; chains contain the maximum number of non-repeated hops in a topology and therefore the highest latency on messages passing from one end of the chain to the other. Chains are highly susceptible to failure due to the fact that any individual link or node failure can disrupt end-to-end communications; the closer the failure is to the root, the more substantial the loss. This is a basic problem with all structures of significant depth.

Using a chain length of 20 nodes, we consider the periodic execution of `cfagent` each minute and measured the time to propagate data from one end of the chain to another, in repeated trials. The result of the completed aggregation for this test is a file on the root node containing each node's CPU load average as well as the time at which that information was collected. Each node used

the cfengine *copy* action to copy a partially aggregated file from its child. Then the node used the cfengine *edit-files* action to append its own load data to the bottom of the file.

The results of the experiments are shown in Figure 4. We shall report on a detailed explanation elsewhere.

The graphs can be understood roughly as follows. The solid line shows a prediction based on the assumption of regular deterministic behaviour. For zero time-delay between receiving and sending in the chain the age of the data is about ten periods. This is what one would expect by random chance: about half the nodes are correctly ordered on average. As the delay is increased to one minute (greater than noise) the noise becomes irrelevant and an optimal number of nodes is correctly ordered for direct transmission. This gives the fastest result. Then as the delay increases, the time increases in steps. If the wait time times the length of the chain is greater than a period, then the nodes on the period boundary will be out of step and will have to wait a whole period to update, hence the jumps in the graph. What is interesting is that the effect of noise is to improve this handicap. There is no room here for a full discussion of this phenomenon, but the result is essential to understand for monitoring.

Promise Trees (Aggregation)

The chain is an unlikely topology in a real distributed system. In most cases one would expect a

node to be able to connect to several other nodes and allow a greater centralization of data during aggregation. We have repeated our experiments for binary trees and the results are shown in Figure 5. The cfengine configuration patterns for these tests are shown in the aggregation examples, Figures 8 and 9.

The data from the tree results are not directly comparable to those of the chain, for several reasons. A number of scales change when performing local aggregation and these changes interfere with the time-scales of system noise. Understanding the tree results is therefore rather more complicated than understanding the chains. The parallel arms of the trees interfere sometimes destructively for parallelized copy and sometimes constructively in serialized copy. Thus our graph seems to reveal a relative stability compared to the chain. This is slightly misleading however. The same basic behaviour is common to both cases; however, the tree is able to delay the onset of temporal instability from chain depth (see [9] for more explanation).

Suppose we put aside the restrictions on topology due to local environment, e.g., the finite range of a wireless network, and ask whether there are reasons for building a tree with a particular number of neighbours (node degree) for aggregation or dissemination. This question should be answered differently depending on who initiates a transmission through the network, how often and at what relative times. In the cfengine model of maintenance in which data are

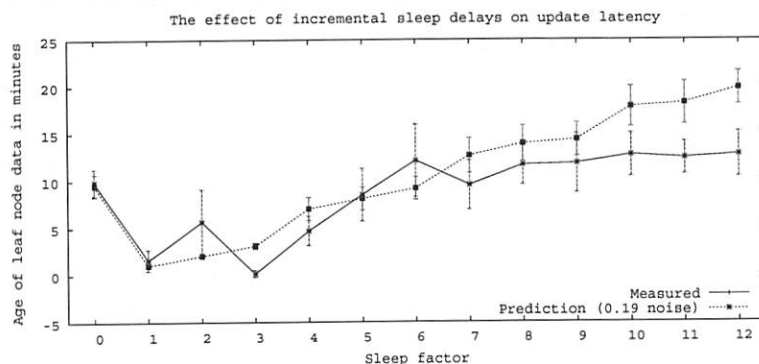


Figure 4: Predicted versus experimental results for the chain propagation. The presence of noise or time-variations actually improves performance compared to a deterministic prediction.

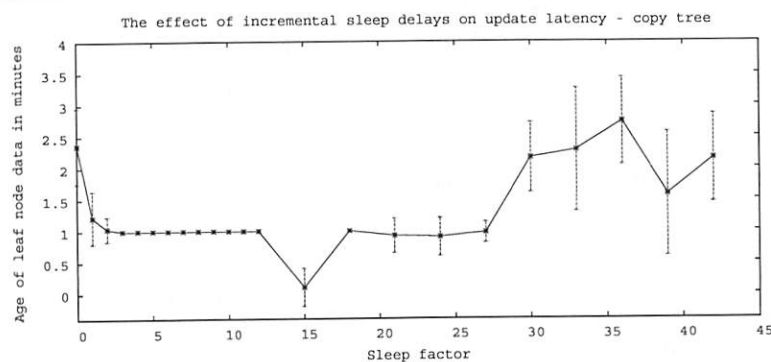


Figure 5: Experimental results for binary tree propagation.

sampled at regular intervals, the behaviour of an aggregation process is something of a cross between the GAP protocol and a Gossip approach [11]. The periodic checking of cfengine promises adds a level of complexity to the data quality of the final result. However, the synchronization of the binary tree is much less sensitive to the size of small offsets than for the chain so it would seem to be advantageous to choose a tree over a chain.

Clearly then the tree is more efficient in terms of time and the decreased network depth gives more freedom in choosing the synchronization parameters. Increasing the node degree (number of children) in the tree increases the processing burden on the aggregator in order to maintain the same accuracy of service level however. A question therefore presents itself: is there an optimal node degree for distributed monitoring?

Scalability

Scalability is about how well a system continues to perform in all its parts as it grows. The burden of size can have a variety of negative effects on a system.

For scalability, we seek to minimize the time to delivery from the leaves of the data structures to the roots (i.e., obtain the lowest value on the vertical axis), while maintaining meaningful data by minimizing temporal fragmentation (partially represented by the error bars). Thus we would like to be as close the lower left of the figures as possible. Our results tell us something about how to achieve this by adjusting overlay topology.

The two structural poles for the network patterns were illustrated in Figure 1: the star pattern for maximum parallelization and centralization (hence maximum burden per root node) and the chain for maximum off-loading and decentralization (hence maximum temporal fragmentation). With centralization, the fraction of the central node's capacity that is available to its children decreases in proportion to the number of clients, so since the capacity is fixed scaling means a reduction of workflow on the children proportional to their own number [1]. In a chain, every node can use its maximum processing capacity on its neighbour and that chain can grow as long as we like until the load of data aggregation (which grows in proportion to its length) becomes a significant burden.

There are thus advantages and disadvantages to each of these structures, with regard to both organization and processing capacity. A tree is essentially a compromise between the two: any tree can be seen as a number of stars chained together. We must decide as a matter of policy what *node degree* or number of branches these stars should have in order to compromise on these two dipolar effects of growth.

One interesting example in which the topology of a network pattern could have a direct effect on scalability is power consumption. Since we envisage network

patterns finding application in mobile ad hoc networks which run off batteries, e.g., sensor networks with limited resources, we should think about the possibility that the choices we make will affect the lifetime of the devices. Power consumption too might have to be traded against speed and accuracy.

We have no generic answer to the question of which kind of structure is best in a given case, as such concerns are a matter for policy. However, consider the following. The rate of power consumption of a node is proportional to its CPU frequency [12] squared. Thus if we design at maximum utilization to cope with demand from aggregation of k neighbours, we must scale cost as k^2 which represents power, cost of cooling or shortened battery life, etc. The risk, on the other hand, associated with not getting data quickly is proportional to the effective depth of the network pattern $(N-1)/k$. So we have a cost function that is a balance between these two

$$Cost = \alpha k^2 + \frac{(N-1)}{k} \quad (4)$$

A plot for this for the arbitrary policy $\alpha = 0.1$ is shown below. This shows the existence of an optimum aggregation degree, in this case $k = 5$. If k were a constant all over the network, i.e., the network formed a regular graph, this would be the optimal answer for minimizing power consumption. However, there are many constraints in ad hoc networks that would make it unlikely to be able to maintain such a regular tree, moreover there are other concerns than power consumption. In general one must compromise between a number of different optimization parameters competing for attention. More detailed considerations then need to be applied to the problem. As we see, the cost rises sharply with increasing centralization, however this does not help roaming hand-held devices with limited range that both cannot centralize and do not want the computational burden focused in one place.

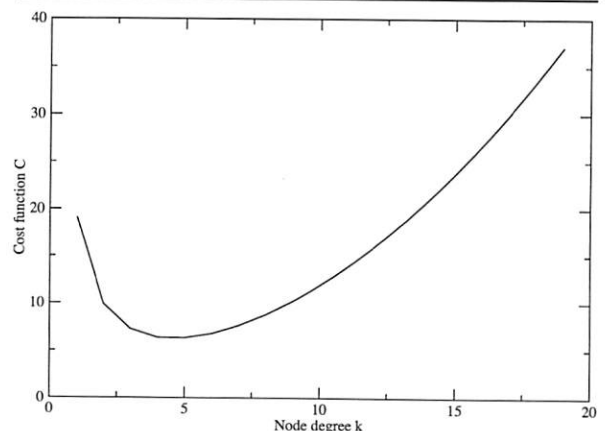


Figure 6: Cost considerations can plausibly lead to an optimum depth of network pattern when power considerations are taken into account. The minimum cost here is given for $k = 5$. Such considerations require an arbitrary choice to be made about relative importance of factors.

Our work here does not offer a simple answer to this conundrum, but shows network managers how to investigate and locate their own compromise as a matter of policy.

Conclusions

In the present work we have provided a proof of concept for implementing network data aggregation and dissemination patterns at the host level, using promise theory inspired methods. We have shown that we can avoid scalability bottlenecks only at the expense of temporal fragmentation of data. If users make logical star networks, they will have the greatest level of certainty about their data but the most fragile architecture in the face of growth. If they choose a number of star topologies chained together they can make a suitable compromise. Most importantly, we point out that the uncertainties incurred should be monitored and presented as part of the data's time-stamps.

We feel that our hybrid network/system study is a stepping stone towards integrating host and network administration within a common framework. Our work has been based on KTH's distributed protocols, and our investigation must be seen as tentative. We have not implemented all the features of the GAP protocol here. The adaptive creation of a network overlay is a topic for a later time, nevertheless some experimental peer to peer features of cfengine are already similar to the ideas used in GAP, and we intend to explore these further. Some partial approximations for this are implemented as *SelectPeerNeighbours*, *SelectPeerLeader* functions in cfengine, with failover options. However, the full details of the algorithm still have to be understood. This will probably take another six months to a year to find the time to complete. Tests are proceeding and will drive a discussion as to the most appropriate way for deciding a topology in a cfengine peer network.

Our microscopic investigation of propagation uncertainty in [9] shows that distributed structures lead to uncertain results. The uncertainties measured in a cfengine network are not simply related to errors in aggregation due to unreliable nodes, as studied in [5, 11], so it is not clear whether the generalization A-GAP would be a realistic solution to the problem here.

The syntax of cfengine's voluntary cooperation model is based on peer to peer interactions, just like promise theory. It was designed with simple one-to-one contracts in mind. We did not consider the possibility of widespread interconnection of contractual relationships. This results in clumsy and cumbersome policy files for encoding patterns in cfengine. Further work is expected to be able to enable regular expressions of some form to more efficiently encode the bilateral promises required for pattern policies.

As we write this, the team at Stockholm has developed a new pattern which they refer to as MGAP, in which every node in a structure can receive a copy of the total aggregate. It seems likely that this pattern

will find a special place in cfengine for extending cfengine's peer to peer monitoring capabilities. We look forward to reporting on this as future work.

This work is supported by the EC IST-EMAN-ICS Network of Excellence (#26854).

Author Biographies

Mark Burgess is professor of Network and System Administration at Oslo University College. He was the first professor with this title. Mark obtained a Ph.D. in Theoretical Physics in Newcastle, for which he received the Runcorn Prize. His current research interests include the behaviour of computers as dynamic systems and applying ideas from physics to describe computer behaviour. Mark is the author of the popular configuration management software package cfengine. He made important contributions to the theory of the field of automation and policy based management, including the idea of operator convergence and promise theory. He is the author of numerous books and papers on Network and System Administration and has won several prizes for his work. Reach him electronically at Mark.Burgess@iu.hio.no.

Matthew Disney has been working in systems administration since 1998. He has a B.S. in Computer Science from the University of Tennessee and an MS in Network and System Administration from the University of Oslo. He is currently working as a cyber security administrator at Oak Ridge National Laboratory.

Rolf Stadler is a professor at the Royal Institute of Technology (KTH) in Stockholm, Sweden, since 2001, where he leads the network management group. He received an M.Sc. degree in mathematics and a Ph.D. in computer science from the University of Zurich, Switzerland, in 1984 and 1990, respectively. Over the last 10 years, Dr. Stadler has been instrumental in the network management research community and served as PC co-chair for premier IEEE conferences in the field, including DSOM'99, NOMS'02, and DSOM'07. He further serves on the editorial board of IEEE Transactions on Network and Service Management (TNSM). His current research interests include scalable networks and systems, autonomous computing, and self management.

Bibliography

- [1] Burgess, M. and G. Canright, "Scalability of Peer Configuration Management in Partially Reliable and Ad Hoc Networks," *Proceedings of the VIII IFIP/IEEE IM Conference on Network Management*, p. 293, 2003.
- [2] Burgess, M. and G. Canright, "Scaling Behaviour of Peer Configuration in Logically Ad Hoc Networks," *IEEE eTransactions on Network and Service Management*, Vol. 1, Num. 1, 2004.
- [3] Burgess, M. and K. Begnum, "Voluntary Cooperation in a Pervasive Computing Environment,"

- Proceedings of the Nineteenth Systems Administration Conference (LISA XIX)*, USENIX Association, Berkeley, CA, p. 143, 2005.
- [4] Lima, K-S and R. Stadler, "A Navigation Pattern for Scalable Internet Management," *Proceedings of the VII IFIP/IEEE IM Conference on Network Management*, 2001.
 - [5] Gonzalez, A., Prieto, and R. Stadler, "Adaptive Distributed Monitoring with Accuracy Objectives," *ACM SIGCOMM Workshop on Internet Network Management (INM 06)*, Pisa, Italy, 2006.
 - [6] Dam, M. and R. Stadler, "A Generic Protocol for Network State Aggregation," *RVK 05*, Linkping, Sweden, June 14-16, 2005.
 - [7] Tel, G., *Introduction to Distributed Algorithms*, Cambridge University Press, 2nd Edition, pp. 181-202, 2000.
 - [8] Chang, E. J. H., "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Transactions on Software Engineering*, Vol. 8, Num. 4, pp. 391-401, 1982.
 - [9] Disney, M., *Exploring Patterns for Scalability of Network Administration with Topology Constraints*, Master's Thesis, Oslo University College, 2007.
 - [10] Burgess, Mark, "An Approach to Understanding Policy Based on Autonomy and Voluntary Cooperation," *IFIP/IEEE 16th International Workshop on Distributed Systems Operations and Management (DSOM)*, LNCS Vol. 3775, pp. 97-108, 2005.
 - [11] Wuhib, F., M. Dam, R. Stadler, and A. Clemm, "Robust Monitoring of Network-Wide Aggregates Through Gossiping," *10th IFIP/IEEE International Symposium on Integrated Management (IM 2007)*, 2007.
 - [12] Burgess, M. and F. Sandnes, "A Promise Theory Approach to Collaborative Power Reduction in a Pervasive Computing Environment," *Springer Lecture Notes in Computer Science*, LNCS Vol. 4159, pp. 615-624, 2006.

Appendix: Examples

```
#####
#
# CHAIN 4 machines 1,2,3,4 (promise chain)
#
#####
classes:
    always = ( any )
    leaf   = ( node4 )
    root   = ( node1 )
#####
control:
    workfile = ( "/tmp/chain-pattern" )
#####
methods:
    #
    # Pattern has to be coded in classes (from)
    # and servers (to)
    #
node1|node2::          # -b | +b - binding
    Aggregate("${workfile}")
        server=node2
        action=method_pattern.cf
        returnvars=ret
        returnclasses=chain_link
node2|node3::
    Aggregate("${workfile}")
        server=node3
        action=method_pattern.cf
        returnvars=ret
        returnclasses=chain_link
node3|node4::
    Aggregate("${workfile}")
        server=node4
        action=method_pattern.cf
        returnvars=ret
        returnclasses=chain_link
#####
editfiles:
    !leaf::
        { $(workfile)
        AutoCreate
        EmptyEntireFilePlease
        AppendIfNoSuchLine "${Aggregate.ret}"
        # Handle errors so no strange loops
        ReplaceAll "Aggregate.ret" With "FAILED"
        }
    leaf::
        { $(workfile)
        AutoCreate
        EmptyEntireFilePlease
        AppendIfNoSuchLine "${value_loadavg}"
        }
#####
alerts:
    root.Aggregate_chain_link::
        "Chain aggregate $(n)$(host)=$(value_loadavg)
        at $(date) $(Aggregate.ret) "
```

Figure 7: A promise chain fully represented as a contract between parties by voluntary cooperation.

```
#####
#
# Netlab config
#
#####
classes:
  leaf    = ( netlab4 )
  root    = ( netlab1 )
#####
control:
  workfile = ( "/tmp/chain-pattern" )
  tempfile = ( "/tmp/chain-temp" )
netlab1::
  serve = ( netlab3 )
netlab3::
  serve = ( netlab4 )
#####
tidy:
#####
copy:
!leaf::
  $(workfile)
  dest=$(tempfile)
  server=$(serve)
  type=checksum
  define=success
  elsedefine=failure
#####
editfiles:
  success::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    InsertFile "$(tempfile)"
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
  failure::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "copy-chain - no response from $(serve)"
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
  leaf::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
#####
alerts:
  success::
    "Chain update succeeded"
    PrintFile("$(workfile)","6")
  failure::
    "No Chain update at $(date)"
```

Figure 8: A simplified version of the promise chain built using a simple pull method. This is much more trusting than the previous example and assumes a certain control over the children.


```
#####
#
# Depth aggregation (promise tree)
#
#####
classes:
  leaf      = ( netlab3 netlab4 )
  aggregator = ( netlab1 )
#####
control:
  workfile = ( "/tmp/chain-pattern" )
  children = (
    A(netlab1,"netlab3,netlab4")
    A(netlab3,"netlab3,netlab4")
    A(netlab4,"netlab3,netlab4")
  )
#####
methods:
netlab1|netlab3|netlab4:: # 2 servers, 1 client
  Aggregate("${workfile}")
    server=$(children[${host}])
    action=method_pattern.cf
    returnvars=ret
    returnclasses=chain_link
#####
editfiles:
  aggregator::
    { ${workfile}
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "${Aggregate_1.ret}"
    AppendIfNoSuchLine "${Aggregate_2.ret}"
    # Handle errors so no strange loops
    ReplaceAll "Aggregate.*ret" With "FAILED"
    }
  leaf::
    { ${workfile}
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "${average_loadavg}"
    }
#####
alerts:
  aggregator.(Aggregate_1_chain_link|Aggregate_2_chain_link)::
    "Chain aggregate ${n}${host}=${average_loadavg} at ${date} \
    ${Aggregate_1.ret} ${Aggregate_2.ret} "
```

Figure 9: A two to one aggregation of text data. This example uses a full promise approach.

```
#####
#
# Breadth aggregation by pull
#
#####
classes:
  leaf    = ( netlab4 netlab3 )
  root    = ( netlab1 )
#####
control:
  Split      = ( , )
  workfile   = ( "/tmp/chain-pattern" )
  tempfile   = ( "/tmp/chain-temp" )
  #
  # One link in a binary tree      1
  #                               / \ aggregation
  #                               3  4
netlab1::
  serve = ( "netlab3,netlab4" )
#####
copy:
!leaf::
  $(workfile)
  dest=$(tempfile)_$(this)
  server=$(serve)
  type=checksum
  define=success
  elsedefine=failure
#####
editfiles:
  success::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    InsertFile "$(tempfile)_$(serve)"
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
  failure::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "copy-chain - no response from $(serve)"
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
  leaf::
    { $(workfile)
    AutoCreate
    EmptyEntireFilePlease
    AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
    }
#####
alerts:
  success::
    "Chain update succeeded"
    PrintFile("$(workfile)","6")
  failure::
    "No Chain update at $(date)"
```

Figure 10: A simpler pull version of the aggregation example.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *login.*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

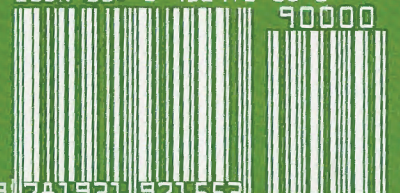
Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

Ajava Systems, Inc.	Hewlett-Packard	rTIN Aps
Cambridge Computer Services, Inc.	IBM	Sendmail, Inc.
cPacket Networks	Infosys	Splunk
DigiCert® SSL Certification	Intel	Sun Microsystems, Inc.
EAGLE Software, Inc.	Interhack	Taos
FOTO SEARCH Stock Footage and Stock Photography	MSB Associates	Tellme Networks
Google	NetApp	UUNET Technologies, Inc.
GroundWork Open Source Solutions	Oracle	VMware
	Raytheon	Zenoss
	Ripe NCC	

ISBN-13: 978-1-931971-55-3
ISBN-10: 1-931971-55-2

90000



9 781931 971553